UNIVERSITÀ DI PARMA

Dipartimento di Scienze Matematiche, Fisiche ed Informatiche

Corso di Laurea in Informatica

Trattamento ed uso del Prodotto Cartesiano tra insiemi tramite la libreria Java JSetL

Applications and uses of Cartesian Product between Sets using the Java library JSetL

Relatore:

Chiar.mo Prof. Gianfranco Rossi

Tesi di Laurea di:

Marco Ghezzi

ANNO ACCADEMICO 2017-2018

Alla mia famiglia, ai miei amici e a coloro che mi hanno sostenuto.

"L'autostrada informatica trasformerà la nostra cultura tanto drasticamente quanto l'invenzione della stampa di Gutenberg ha trasformato quella del Medio Evo."

Bill Gates

Indice

Capitolo 1

Introduzione

I prodotti Cartesiani sono un concetto matematico ricorrente nel mondo dell'informatica, utilizzato principalmente con lo scopo di gestire una particolare struttura di dati. In [1], Maximiliano Cristià e Gianfranco Rossi hanno definito un linguaggio per la programmazione logica con vincoli (CLP), denominato $\mathcal{L}_{\mathcal{BR}}$, che estende il linguaggio logico $\texttt{CLP}(\mathcal{SET})$[2] offrendo la possibilità di definire e manipolare le relazioni binarie. Le formule in $\mathcal{L}_{\mathcal{BR}}$ sono formule del primo ordine prive di quantificatori, che ammettono la maggior parte dei classici operatori relazionali del primo ordine, come dominio, composizione, etc. $\mathcal{L}_{\mathcal{BR}}$ offre anche un risolutore (solver), denominato $SAT_{\mathcal{BR}}$, tramite cui è possibile verificare la soddisfacibilità delle formule esprimibili nel linguaggio stesso. Il linguaggio e il risolutore sono implementati all'interno di un sistema prolog denominato {log}[3].
$\mathcal{L}_{\mathcal{BR}}$vanta tre importanti proprietà:

(a) Può risolvere formule senza dover specificare dei domini finiti per le variabili presenti nelle formule;

(b) Può calcolare una rappresentazione finita di tutte le (infinite) soluzioni di una data formula;

(c) Insiemi e relazioni binarie possono essere combinate liberamente.

In [4], gli stessi autori hanno definito un'ulteriore estensione di $\mathcal{L}_{\mathcal{BR}}$, chiamata $\mathcal{L}_{\mathcal{CP}}$, in cui il prodotto Cartesiano diventa un'entità di prima classe. Facendo questo, il prodotto tra due insiemi ammissibili viene espresso tramite un nuovo termine di insieme che può essere manipolato tramite i vincoli insiemistici già presenti in $\mathcal{L}_{\mathcal{BR}}$ opportunamente adattati.

Questo nuovo tipo di insieme può essere liberamente combinato con tutti gli altri tipi di insiemi disponibili nel linguaggio. L'introduzione del prodotto Cartesiano permette, tra l'altro, semplici definizioni di complementazione e diversità. In questo modo, $\mathcal{L}_{\mathcal{CP}}$ consente di esprimere tutte le formule dell'algebra delle relazioni su insiemi di relazioni finite in un universo numerabile.

JSetL [5; 6] è una libreria Java che fornisce un'implementazione pressochè completa del linguaggio `CLP(`$\mathcal{SET}$`)` all'interno di Java. In particolare JSetL offre un'implementazione Java di concetti tipici dei linguaggi CLP, come variabili logiche, liste, unificazione, risoluzione di vincoli e non determinismo. Recentemente [7] è stata estesa in modo da implementare il linguaggio $\mathcal{L}_{\mathcal{BR}}$ al suo interno offrendo, in particolare, classi e metodi per realizzare e manipolare (tramite vincoli) relazioni binarie e funzioni parziali.

Lo scopo di questo lavoro di tesi è quello di progettare e realizzare un'ulteriore estensione dell'attuale libreria JSetL, introducendo in essa la possibilità di definire e manipolare i prodotti Cartesiani, così come supportati dal linguaggio $\mathcal{L}_{\mathcal{CP}}$. In particolare, sfruttando il meccanismo dell'ereditarietà offerto da Java, verrà creata la classe `CP`, per l'implementazione del prodotto Cartesiano, come estensione della classe `LReL` (classe che definisce la relazioni binarie). Verranno inoltre modificate ed estese le classi `RwRulesEq` e `RwRulesSet` per contenere le implementazioni delle regole di riscrittura dei vincoli che operano sui prodotti Cartesiani.

La tesi è organizzata nel seguente modo:

- Il Capitolo 2 introduce il linguaggio {`log`} e la procedura di risoluzione adottata dal risolutore, seguita da una presentazione del linguaggio $\mathcal{L}_{\mathcal{CP}}$ sui prodotti Cartesiani con la relativa sintassi e semantica propria e dei nuovi vincoli aggiunti.

- Il Capitolo 3 tratta le regole di riscrittura dei nuovi vincoli per i prodotti Cartesiani (CP), mostrando come questi nuovi oggetti si comportano in relazione a quelli già presenti, fornendo anche alcune tabelle di copertura che mettono in evidenza quale regola viene richiamata durante una specifica interazione con un CP.

- Il Capitolo 4 fa un breve richiamo alle classi presenti all'interno della libreria JSetL, successivamente viene presentata la definizione della nuova classe `CP` e dei suoi metodi, che farà da riferimento per l'implementazione del linguaggio $\mathcal{L}_{\mathcal{CP}}$ all'interno di JSetL.

- Il Capitolo 5 mostra come le regole di riscrittura che operano sugli oggetti di classe `CP` sono state implementate all'interno della libreria di JSetL, andando a vedere il codice nel dettaglio con alcuni esempi di utilizzo dei vari vincoli estesi.

- Il Capitolo 6 presenta alcune considerazioni sul lavoro svolto e accenna ad eventuali lavori e sviluppi futuri da affrontare.

Capitolo 2

Visione logica dei Prodotti Cartesiani

Il capitolo inizia con un'introduzione informale del linguaggio $\mathcal{L}_{\mathcal{BR}}$ implementato all'interno di {log}, presentando in particolare i vincoli di base presenti e il loro funzionamento, per poi parlare della sua estensione che supporta i prodotti Cartesiani, ovvero il linguaggio $\mathcal{L}_{\mathcal{CP}}$.

2.1 Introduzione informale al linguaggio {log}

{log} è un linguaggio di programmazione logica a vincoli e un risolutore di soddisfacibilità per un frammento della teoria degli insiemi finiti e per le relazioni binarie.
In questo capitolo verrà presentata una breve introduzione di questo particolare linguaggio, enfatizzando alcune sue importanti proprietà.
Una prima proprietà di {log}è che le relazioni binarie sono insiemi di coppie ordinate, quindi possono essere liberamente combinate con altri insiemi, e gli operatori insiemistici possono prendere le relazioni come argomenti. Per esempio, la seguente è una possibile formula in {log}:

$$un(A, B, \{(1, 2), (h, 3) \sqcup C\}) \wedge dom(A, \{1, 2\}) \wedge dom(B, X) \wedge 1 \notin X$$

dove:

- A, B, C e X sono variabili insiemistiche e h è una variabile di tipo qualsiasi.
- Il termine insiemistico $\{(1,2),(h,3) \sqcup C\}$ viene detto *insieme estensionale*, ed è interpretato come $\{(1,2),(h,3)\} \cup C$ dove $(1,2)$ e $(h,3)$ rappresentano delle coppie.
- $un(A, B, \{(1,2),(h,3) \sqcup C\})$ afferma che $\{(1,2),(h,3) \sqcup C\} = A \cup B$.
- $dom(A, \{1,2\})$ afferma che $\{1,2\}$ è il dominio di A.

Gli insiemi estensionali sono entità di prima classe e costituiscono la base di tutto il linguaggio.
In $\{x \sqcup A\}$, x viene detta *parte elemento* e A *parte insiemistica*. Scritture nella forma $\{x \sqcup \{y \sqcup A\}\}$ vengono abbreviate come $\{x, y \sqcup A\}$, e se A è un insieme vuoto ($\emptyset$) viene semplicemente omesso $\{x, y\}$.
Gli insiemi estensionali possiedono le due proprietà elementari degli insiemi, ovvero l'assorbimento ($\{a, a\} = \{a\}$) e la commutazione sulla sinistra ($\{a, b \sqcup A\} = \{b, a \sqcup A\}$). Inoltre, l'equivalenza tra due insiemi estensionali è governata dall'unificazione insiemistica. Gli insiemi estensionali sono sempre finiti e senza tipo, inoltre essi possono essere a loro volta elementi di un insieme (insiemi annidati).

Un'altra proprietà importante di {`log`}è che gli insiemi estensionali (e di conseguenza anche le relazioni binarie) possono essere *parzialmente specificate*. Questo implica che, sia gli elementi che le parti insiemistiche possono essere variabili. Notare che la cardinalità di un insieme la cui parte insiemistica è variabile risulta non specificato (diventa un insieme finito non limitato poichè può avere qualsiasi numero finito di elementi). Questa proprietà si rapporta molto bene con l'abilità di {`log`} di decidere la soddisfacibilità di formule senza dover necessariamente specificare un dominio finito per ogni variabile di insieme.

Una terza proprietà importante è che {log} produce una rappresentazione finita di tutte le soluzioni di una data formula soddisfacibile, anche se esse risultano essere infinite. In questo contesto, una 'soluzione' è un assegnamento di valori a tutte le variabili di tale formula. Per esempio, una formula come quella riportata precedentemente ha un numero infinito di soluzioni, tuttavia {log} è in grado di calcolare una rappresentazione finita per tutte queste soluzioni nella forma di una disgiunzione finita di formule la cui correttezza è garantita.

Per quanto riguarda l'espressività del linguaggio $\mathcal{L}_{\mathcal{BR}}$, esso supporta diversi operatori relazionali e insiemistici. In particolare, fornisce procedure di riscrittura per i vincoli di uguaglianza, appartenenza, unione, disgiunzione, composizione, identità, inversa e tutte le loro negazioni. Questi vincoli sono detti *vincoli di base*. Attraverso loro molti altri operatori possono essere definiti all'interno del linguaggio, come semplici congiunzioni e disgiunzioni di vincoli di base (detti *vincoli definiti*).
I prodotti Cartesiani non sono entità di prima classe in $\mathcal{L}_{\mathcal{BR}}$. In realtà in $\mathcal{L}_{\mathcal{BR}}$ sarebbe possibile definire un nuovo vincolo derivato in cui il prodotto Cartesiano di due insiemi A e B viene espresso tramite una congiunzione (non banale) di vincoli primitivi su relazioni e insiemi. Sebbene possibile da un punto di vista teorico, questo approccio si è subito rivelato poco realistico in pratica, dato che il trattamento di formule che coinvolgono prodotti Cartesiani espressi in questo modo diventa spesso eccessivamente inefficiente.
Per questo motivo, in [Cristià Rossi 2018] è stato definito un nuovo linguaggio logico, denominato $\mathcal{L}_{\mathcal{CP}}$, con i seguenti obiettivi:

(a) $\mathcal{L}_{\mathcal{CP}}$ estende $\mathcal{L}_{\mathcal{BR}}$con la definizione di prodotto Cartesiano come un nuovo termine insiemistico;

(b) la maggior parte delle procedure di riscrittura degli operatori di base presenti in $\mathcal{L}_{\mathcal{BR}}$ sono state estese in modo che possano supportare il nuovo tipo di insieme;

(c) si è dimostrato che le estensioni delle riscritture sono complete;

(d) queste estensioni sono state implementate in {log};

(e) è stata condotta una completa valutazione empirica dell'implementazione che ne ha mostrato la validità pratica.

2.2 Il linguaggio $\mathcal{L}_{\mathcal{CP}}$

Il linguaggio del prodotto Cartesiano, $\mathcal{L}_{\mathcal{CP}}$, è un'estensione del linguaggio delle relazioni binarie $\mathcal{L}_{\mathcal{BR}}$. Informalmente, $\mathcal{L}_{\mathcal{CP}}$ estende $\mathcal{L}_{\mathcal{BR}}$ definendo un nuovo termine insiemistico, il prodotto Cartesiano tra due insiemi, e permettendo agli operatori insiemistici di operare allo stesso modo anche su questo nuovo tipo di insieme.

Nota. Questo progetto di estensione del linguaggio $\mathcal{L}_{\mathcal{BR}}$ è tutt'ora in sviluppo, di conseguenza al momento sono state implementate solamente estensioni per i vincoli di uguaglianza, appartenenza, unione, disgiunzione, con le relative negazioni e solamente queste verranno trattate in questo documento.

2.2.1 Sintassi

La sintassi del linguaggio $\mathcal{L}_{\mathcal{CP}}$ viene definita fornendo le specifiche su cui si costruiscono termini e formule del linguaggio.

Il linguaggio $\mathcal{L}_{\mathcal{CP}}$ è costruito mediante i seguenti insiemi di simboli:

1. $\mathcal{F}$ è l'insieme dei simboli di funzione, il quale viene suddiviso a sua volta in due sottoinsiemi: il primo, detto $\mathcal{F}_{\mathcal{S}}$, contiene $\emptyset$, $\{\cdot \sqcup \cdot\}$ e $\{\cdot \times \cdot\}$, mentre il secondo, denominato $\mathcal{F}_{\mathcal{X}}$ contiene al suo interno almeno il simbolo di funzione $\{\cdot, \cdot\}$;

2. Π è l'insieme dei simboli primitivi degli operatori insiemistici, ovvero $=, \in, set, rel, dom, inv, comp, un, \mid\mid, pfun$;

3. $\mathcal{V}$ è un insieme numerabile di variabili diviso in due: $\mathcal{V}_\mathcal{S}$ e $\mathcal{V}_\mathcal{X}$

Intuitivamente, $\emptyset$ rappresenta l'insieme vuoto, $\{x \sqcup X\}$ rappresenta l'insieme $\{x\} \cup X$ e $X \times Y$ rappresenta il prodotto Cartesiano tra due insiemi X e Y, secondo la regola $X \times Y = \{(x, y) | x \in X \wedge y \in Y\}$. Inoltre, (x, y) rappresenta una coppia ordinata.
I termini CP, rappresentanti un prodotto Cartesiano, sono costruiti usando i simboli presenti in $\mathcal{F}$ e $\mathcal{V}$ come segue.

Definizione: Termini CP. L'insieme dei termini CP è il sottoinsieme massimo dell'insieme dei termini generati dalla seguente grammatica:

$$\mathcal{T}^0{}_{\mathcal{CP}} ::= Elem | Set \qquad Elem ::= \mathcal{T}_\mathcal{X} | \mathcal{V}$$
$$Set ::= '\emptyset' \mid '\}' \; \mathcal{T}^0{}_{\mathcal{CP}} \; ' \sqcup' \; Set \; '\}' \mid Set \; '\times' \; Set | \mathcal{V}_\mathcal{S}$$

dove $\mathcal{V}$ ($\mathcal{V}_\mathcal{S}$) rappresenta qualsiasi variabile in $\mathcal{V}$ (dell'insieme Set), mentre $\mathcal{T}_\mathcal{X}$ rappresenta l'insieme dei termini $\mathcal{X}$ non variabili costruiti usando li simboli in $\mathcal{F}_\mathcal{X}$.

I termini dell'insieme Set sono chiamati *termini di insieme*; in particolare, i termini di insieme della forma $\cdot \times \cdot$ sono termini di *prodotto Cartesiano* o semplicemente termini *prodotto*. Da notare che, essendo termini di insieme, i prodotti Cartesiani possono essere combinati con altri termini di insieme. Quindi, è possibile scrivere i termini come $X \times (Y \times Z)$, $\{x \sqcup X \times Y\}$, etc.

Definizione: Vincoli CP. Un vincolo CP è qualsiasi predicato atomico costituito da simboli presenti in Π. L'insieme dei vincoli CP è denotato come $\mathcal{C}_\mathcal{CP}$.
Definizione: Formule CP. l'insieme delle formule CP, denotato come $\Phi_\mathcal{CP}$, è dato come: $\Phi_\mathcal{CP} ::= true \mid C_\mathcal{CP} \mid \neg C_\mathcal{CP} \mid \Phi_\mathcal{CP} \wedge \Phi_\mathcal{CP} \mid \Phi_\mathcal{CP} \vee \Phi_\mathcal{CP}$.

2.2.2 Semantica

La sintassi viene interpretata seguendo la struttura di interpretazione $\mathcal{R} = \left\langle D, (\cdot)^{\mathcal{R}} \right\rangle$, dove D e $(\cdot)^{\mathcal{R}}$ sono definite nel seguente modo.

Definizione: Dominio di interpretazione. Il dominio di interpretazione D è suddiviso in due parti: D_{Set} e D_{X}, dove D_{Set} indica la collezione di tutti gli insiemi finiti costruiti a partire dagli elementi in D_{X}, mentre D_{X} è una collezione di qualsiasi altro oggetto che non fa parte di D_{Set}.

Definizione: Funzione di interpretazione. La funzione di interpretazione $(\cdot)^{\mathcal{R}}$ per i simboli di predicato in Φ si definisce come: $x = y$ viene interpretato come $x^{\mathcal{R}} = y^{\mathcal{R}}$; $x \in X$ come $x^{\mathcal{R}} \in X^{\mathcal{R}}$; $x \mid\mid X$ come $x^{\mathcal{R}} \mid\mid X^{\mathcal{R}}$; $comp(A, B, C)$ come $C^{\mathcal{R}} = A^{\mathcal{R}} \odot B^{\mathcal{R}}$, dove $\odot$ è l'operatore di composizione in D_{Set}; e $un(X, Y, Z)$ come $Z^{\mathcal{R}} = X^{\mathcal{R}} \cup Y^{\mathcal{R}}$.

Capitolo 3

Regole di riscrittura dei CP

In questo capitolo verrà mostrato come le procedure di riscrittura presenti in $\mathcal{L}_{\mathcal{CP}}$ per uguaglianza, appartenenza, disgiunzione, unione, con le relative negazioni, sono estese per accogliere i prodotti Cartesiani.
La procedura proposta sfrutta l'unificazione insiemistica per far fronte all'uguaglianza tra i termini insiemistici. Il risolutore $SAT_{\mathcal{BR}}$, lo strumento dedicato alla risoluzione dei vincoli, applica procedure di riscrittura specializzate alla formula corrente Φ e ritorna la formula modificata. Ogni procedura di riscrittura applica qualche regola non deterministica per ridurre la complessità sintattica dei vincoli di un certo tipo. Quando nessuna regola è applicabile per una data formula, la procedura di riscrittura termina immediatamente e la formula resta inalterata. Questo processo si ripete finchè un dato punto di controllo viene raggiunto, ovvero quando la formula non può essere ulteriormente semplificata, dopo di che le formule atomiche irriducibili sono ritornate come parte della risposta calcolata.

Il capitolo riporta tutte le regole di riscrittura relative ai prodotti Catesiani, secondo la seguente notazione:

- t, u, d indicano termini qualsiasi;
- A, B, C, D, E, Z rappresentano un qualsiasi insieme (estensionale o CP);

- $\bar{A}, \bar{B}, \bar{C}$ sono insiemi variabili o cp variabili;
- W, X, Y indicano insiemi variabili;
- n, a, b, c, x, y sono elementi generici;
- z indica tipicamente una coppia di elementi n_1 e n_2;
- $\emptyset$ rappresenta un insieme vuoto o un CP dove almeno uno dei due insiemi da cui è composto vuoto.

3.1 Regole di Uguaglianza $(=)$

1. $\emptyset = \emptyset \rightarrow true$
2. $\bar{A} = \bar{A} \rightarrow true$
3. Se A non è una variabile
   $A = X \rightarrow X = A$
4. If $X \in vars(t_0, ..., t_n)$:
   $X = \{t_0, ..., t_n \sqcup A\} \rightarrow false$
5. If $X \notin vars(t_0, ..., t_n)$:
   $X = \{t_0, ..., t_n \sqcup X\} \rightarrow X = \{t_0, ..., t_n \sqcup N\}$
6. Se X è presente in altri vincoli dell'espressione in input
   $X = A \rightarrow$ sostituisce X con A nel resto dell'espressione.
7. $\{t \sqcup A\} = \emptyset \rightarrow false$
8. $\emptyset = \{t \sqcup A\} \rightarrow false$
9. $\{t_0, ..., t_m \sqcup X\} = \{u_0, ..., u_n \sqcup X\} \rightarrow$
   $(t_0 = u_j \wedge \{t_1, ..., t_m \sqcup X\} = \{u_0, ..., u_{j-1}, u_{j+1}, ..., u_n \sqcup X\}$
   $\vee t_0 = u_j \wedge \{t_0, ..., t_m \sqcup X\} = \{u_0, ..., u_{j-1}, u_{j+1}, ..., u_n \sqcup X\}$
   $\vee t_0 = u_j \wedge \{t_1, ..., t_m \sqcup X\} = \{u_0, ..., u_n \sqcup X\}$
   $\vee X = \{t_0 \sqcup N\} \wedge \{t_1, ..., t_m \sqcup N\} = \{u_0, ..., u_n \sqcup N\})$

10. $\{t \sqcup A\} = \{u \sqcup B\} \rightarrow$
    $t = u \wedge A = B \vee t = u \wedge \{t \sqcup A\} = B \vee t = u \wedge A = \{u \sqcup B\} \vee A =$
    $\{u \sqcup N\} \wedge \{t \sqcup N\} = B$

11. $X \times Y = \emptyset \rightarrow X = \emptyset \vee Y = \emptyset$

12. $\emptyset = X \times Y \rightarrow X = \emptyset \vee Y = \emptyset$

13. $A \times B = C \times D \rightarrow$
    $(A = C \wedge B = D \wedge A \times B \neq \emptyset \wedge C \times D \neq \emptyset)$
    $\vee (A \times B = \emptyset \wedge C \times D = \emptyset)$

14. $X \times Y = \{z_1, ..., z_k \sqcup X \times Y\} = \bigwedge_{i=1}^{k} z_i \in X \times Y$

15. $\{z_1, ..., z_k \sqcup X \times Y\} = X \times Y = \bigwedge_{i=1}^{k} z_i \in X \times Y$

16. $A \times B = \{z_1 \sqcup Z_1\} \rightarrow A = \{n_1 \sqcup N_1\} \wedge B = \{n_2 \sqcup N_2\} \wedge z = (n_1, n_2)$
    $\wedge\, un(\{n_1\} \times N_2, N_1 \times \{\{n_1\} \sqcup N_2\}, Z)$

17. $\{z_1 \sqcup Z_1\} = A \times B \rightarrow A = \{n_1 \sqcup N_1\} \wedge B = \{n_2 \sqcup N_2\} \wedge z = (n_1, n_2)$
    $\wedge\, un(\{n_1\} \times N_2, N_1 \times \{\{n_1\} \sqcup N_2\}, Z)$

Solved forms

1. $X = t$ e nè t e nemmeno le altre formule contengono X

3.2 Regole di Disuguaglianza ($\neq$)

1. $\emptyset \neq \emptyset \rightarrow false$
2. $\bar{A} \neq \bar{A} \rightarrow false$
3. Se A non è nè variabile nè CP
   $A \neq X \rightarrow X \neq A$
4. Se $X \notin vars(t_1, ..., t_n)$:
   $X \neq \{t_1, ..., t_n \sqcup X\} \rightarrow (t_1 \notin X \vee ... \vee t_n \notin X)$
5. Se $X \in vars(t_1, ..., t_n)$:
   $X \neq \{t_1, ..., t_n \sqcup A\} \rightarrow true$
6. $\emptyset \neq \{t \sqcup A\} \rightarrow true$
7. $\{t \sqcup A\} \neq \emptyset \rightarrow true$
8. $\{t \sqcup A\} \neq \{u \sqcup B\} \rightarrow$
   $(y \in \{t \sqcup A\} \wedge y \notin \{u \sqcup B\}) \vee (y \notin \{t \sqcup A\} \wedge y \in \{u \sqcup B\})$
9. $A \times B \neq Z \rightarrow (n \in A \times B \wedge n \notin Z) \vee (n \notin A \times B \wedge n \in Z)$
10. $Z \neq A \times B \rightarrow (n \in A \times B \wedge n \notin Z) \vee (n \notin A \times B \wedge n \in Z)$

Solved forms

1. $X \neq t$, dove t non è un prodotto, e X non compare nè in t nè come argomento di alcun predicato $p(\cdots), p \in un, dom, inv, comp$

3.3 Regole di Appartenenza ($\in$)

1. $t \in \emptyset \rightarrow false$
2. $t \in \{u \sqcup A\} \rightarrow t = u \vee t \in A$
3. $t \in X \rightarrow X = \{t \sqcup N\}$
4. $z \in A \times B \rightarrow z = (n_1, n_2) \wedge n_1 \in A \wedge n_2 \in B$

Solved forms

Nessuna

3.4 Regole di Non Appartenenza ($\notin$)

1. $t \notin \emptyset \rightarrow true$
2. $t \notin \{u \sqcup A\} \rightarrow t \neq u \wedge t \notin A$
3. If $X \in vars(t)$:
   $t \notin X \rightarrow true$
4. $z \notin A \times B \rightarrow z = (n_1, n_2) \wedge (n_1 \notin A \vee n_2 \notin B)$

Solved forms

1. $t \notin X$ e X non compare in t

3.5 Regole di Disgiunzione ($||$)

1. $\bar{A} \mid\mid \bar{A} \rightarrow \bar{A} = \emptyset$
2. $A \mid\mid \emptyset \rightarrow true$
3. $\emptyset \mid\mid A \rightarrow true$
4. $A \mid\mid \{t \sqcup B\} \rightarrow t \notin A \wedge A \mid\mid B$
5. $\{t \sqcup B\} \mid\mid A \rightarrow t \notin A \wedge A \mid\mid B$
6. If $Z \not\equiv \emptyset$:
   $\{a \sqcup A\} \times \{b \sqcup B\} \mid\mid Z \rightarrow \{(a, b) \sqcup N\} \mid\mid Z \wedge un(a \times B, A \times \{b \sqcup B\}, N)$
7. If $Z \not\equiv \emptyset$:
   $Z \mid\mid \{a \sqcup A\} \times \{b \sqcup B\} \rightarrow \{(a, b) \sqcup N\} \mid\mid Z \wedge un(a \times B, A \times \{b \sqcup B\}, N)$

Solved forms

1. $\bar{A} \mid\mid \bar{B}$ dove $\bar{A} \not\equiv \bar{B}$

3.6 Regole di Non Disgiunzione ($\not||$)

1. $A \times B \not|| Z \rightarrow n \in A \times B \wedge n \in Z$

Solved forms

Nessuna

3.7 Regole di Unione $(\cup)$

1. $un(\bar{A}, \bar{A}, B) \rightarrow \bar{A} = B$
2. $un(A, B, \emptyset) \rightarrow A = \emptyset \wedge B = \emptyset$
3. $un(\emptyset, A, \bar{B}) \rightarrow \bar{B} = A$
4. $un(A, \emptyset, \bar{B}) \rightarrow \bar{B} = A$
5. $un(\{t \sqcup C\}, A, \bar{B}) \rightarrow$
   $(t \notin A \wedge un(N_1, A, N)$
   $\vee A = \{t \sqcup N_2\} \wedge un(N_1, N_2, N))$
   $\wedge \{t \sqcup C\} = \{t \sqcup N_1\} \wedge \bar{B} = \{t \sqcup N\}$
6. $un(A, \{t \sqcup C\}, \bar{B}) \rightarrow$
   $(t \notin A \wedge un(N_1, A, N)$
   $\vee A = \{t \sqcup N_2\} \wedge un(N_1, N_2, N))$
   $\wedge \{t \sqcup C\} = \{t \sqcup N_1\} \wedge \bar{B} = \{t \sqcup N\}$
7. $un(A, B, \{t \sqcup C\}) \rightarrow$
   $(A = \{t \sqcup N_1\} \wedge un(N_1, B, N)$
   $\vee B = \{t \sqcup N_1\} \wedge un(A, N_1, N)$
   $\vee A = \{t \sqcup N_1\} \wedge B = \{t \sqcup N_2\} \wedge un(N_1, N_2, N))$
   $\wedge \{t \sqcup C\} = \{t \sqcup N\}$
8. Se almeno uno tra X, Y, Z è un prodotto non variabile:
   $un(A, B, C) \rightarrow \mathcal{U}(A) \wedge \mathcal{U}(B) \wedge \mathcal{U}(C) \wedge un(\mathcal{T}(A), \mathcal{T}(B), \mathcal{T}(C))$

La funzione $\mathcal{T}$ è definita come:

$$\mathcal{T}(t) = \begin{cases} \{(a_i, b_i) \sqcup N_i\} & \text{if } t \equiv \{a_i \sqcup A_i\} \times \{b_i \sqcup B_i\} \\ \emptyset & \text{if } t \equiv \emptyset \times B_i \text{ or } t \equiv A_i \times \emptyset \\ t & \text{altrimenti} \end{cases} \tag{3.1}$$

La funzione $\mathcal{U}$ è definita come:

$$\mathcal{U}(t) = \begin{cases} un(\{a_i\} \times B_i, A_i \times \{b_i \sqcup B_i\}, N_i) & \text{if } t \equiv \{a_i \sqcup A_i\} \times \{b_i \sqcup B_i\} \wedge A_i \not\equiv \emptyset \wedge B_i \not\equiv \emptyset \\ N_i = A_i \times \{b_i\} & \text{if } t \equiv \{a_i \sqcup A_i\} \times \{b_i \sqcup \emptyset\} \wedge A_i \not\equiv \emptyset \\ N_i = \{a_i\} \times B_i & \text{if } t \equiv \{a_i \sqcup \emptyset\} \times \{b_i \sqcup B_i\} \wedge B_i \not\equiv \emptyset \\ N_i = \emptyset & \text{if } t \equiv \{a_i \sqcup \emptyset\} \times \{b_i \sqcup \emptyset\} \\ true & \text{altrimenti} \end{cases} \tag{3.2}$$

Solved forms

1. $un(\bar{A}, \bar{B}, \bar{C})$ con $\bar{A} \not\equiv \bar{B}$

3.8 Tabelle di copertura delle regole

Dopo aver definito tutte le regole di riscrittura per i CP, per chiarezza sono state costruite delle tabelle per aiutare a capire quale regola si applica in una determinata situazione. Le tabelle si leggono prendendo come primo argomento uno presente sulla colonna a sinistra, l'operatore è specificato nell'angolo in alto a sinistra, mentre il secondo argomento è scelto nella prima riga in alto. Per esempio trovandosi di fronte $1, 2 \times 3, 4 = \emptyset$ si prenderà in considerazione la regola numero 11 dell'uguaglianza.
Gli argomenti hanno il seguente significato:

- $\emptyset$ rappresenta, come già detto precedentemente, un insieme vuoto o un CP con almeno uno dei due insiemi che lo compongono vuoto.
- **Init-Set** rappresenta un insieme inizializzato,
- **Var-Set** rappresenta un insieme variabile,
- **CP** rappresenta un prodotto Cartesiano qualsiasi,
- **Init-CP** rappresenta un prodotto Cartesiano inizializzato,

- **Var-Set** rappresenta un prodotto Cartesiano variabile,

$=$	$\emptyset$	**Init-Set**	**Var-Set**	**CP**
$\emptyset$	1	8	3	12
Init-Set	7	10 (9)	3	17(15)
Var-Set	6(SF1)	6 (4,5,SF1)	6(2,SF1)	6(SF1)
CP	11	16 (14)	3	13(2)

$\neq$	$\emptyset$	**Init-Set**	**Var-Set**	**CP**
$\emptyset$	1	6	3	10
Init-Set	7	8	3	10
Var-Set	SF1	SF1(4,5)	SF1(2)	10
CP	9	9	9	9(2)

$\in$	$\emptyset$	**Init-Set**	**Var-Set**	**CP**
$\emptyset$	1	2	3	4

$\notin$	$\emptyset$	**Init-Set**	**Var-Set**	**CP**
$\emptyset$	1	2	SF1(3)	4

$\|\|$	$\emptyset$	**Init-Set**	**Var-Set**	**Init-CP**	**Var-CP**
$\emptyset$	2	3	3	3	3
Init-Set	2	4	5	5	5
Var-Set	2	4	SF1(1)	7	SF1
Init-CP	2	4	6	6	6
Var-CP	2	4	SF1	7	SF1(1)

union	All Cases
$un(A, B, \emptyset)$	2(1)
$un(A, B, \{c \sqcup C\})$	7(1)
$un(A, B, \{c \sqcup C\} \times \{d \sqcup D\})$	8(1)

$un(A, B, \bar{C})$	$\emptyset$	**Init-Set**	**Var-Set**	**Init-CP**	**Var-CP**
$\emptyset$	3	3	3	3	3
Init-Set	4	5	5	5	5
Var-Set	4	6	SF1(1)	8	SF1
Init-CP	4	6	8	8	8
Var-CP	4	6	SF1	8	SF1(1)

Capitolo 4

I Prodotti Cartesiani in JSetL: Implementazione di $\mathcal{L}_{\mathcal{CP}}$

JSetL è una libreria Java utilizzata per il supporto della programmazione dichiarativa nel linguaggio imperativo Object Oriented Java.
L'attuale versione offre un'implementazione pressochè completa del linguaggio $\mathcal{L}_{\mathcal{BR}}$ tramite opportune classi e metodi di libreria. In questo capitolo verranno dapprima presentate le classi principali della libreria, quali LVar, LSet, Constraint e SolverClass, ovvero quelle più importanti agli scopi di questo lavoro di tesi. Successivamente verrà mostrato come estendere l'implementazione attuale di JSetL in modo da supportare il linguaggio dei prodotti Cartesiani $\mathcal{L}_{\mathcal{CP}}$.

4.1 La libreria JSetL

4.1.1 La classe `LVar`

La classe `LVar` permette l'implementazione delle variabili logiche. Permette di creare oggetti in grado di contenere come loro valore qualsiasi oggetto Java (ad esempio Integer e String).
Si possono creare variabili logiche con un valore già assegnato in partenza (inizializzate o bound) o prive di valore (non inizializzate o unbound); inoltre,

è possibile assegnare un nome esterno (di fatto una stringa) a ciascuna di esse.

Di seguito si mostrano alcuni esempi relativi alla creazione di un `LVar`:

```
// Creazione di una variabile logica non inizializzata e senza nome
LVar var1 = new LVar();
// Creazione di una variabile logica non inizializzata di nome x
LVar var2 = new LVar("x");
// Creazione una variabile logica contenente un oggetto integer
// e senza nome
LVar var3 = new LVar(256);
```

Gli oggetti `LVar` supportano i vincoli di uguaglianza (`eq`), disuguaglianza (`neq`), appartenenza (`in`) e non appartenenza (`nin`).

4.1.2 La classe `LSet`

La classe `LSet` realizza l'implementazione di insiemi logici formati da oggetti qualsiasi, anche da altri `LSet`.
Come per gli `LVar`, si possono creare insiemi logici con un valore già assegnato in partenza (inizializzate o bound) o privi di valore (non inizializzate o unbound); inoltre, è possibile assegnare un nome esterno a ciascuno di essi.
Un insieme logico può essere limitato e definito (o chiuso) come $\{1, a, \{\}, \{a, b\}\}$, oppure illimitato parzialmente definito (o aperto) come $\{1, a, \{\}, \{a, b\} \mid R\}$ con $R \in \mathcal{V}$.

Di seguito si mostrano alcuni esempi relativi alla creazione di un `LSet`:

```
// Creazione di un insieme logico non inizializzato e senza nome
LSet set1 = new LSet();
// Creazione di un insieme logico non inizializzato di nome X
LSet set2 = new LSet("X");
// Creazione di un insieme logico definito {1,a,{}} con nome S
LSet set3 = LSet.empty().ins(1).ins(a).ins(LSet.empty()).setName("S");
// Creazione di un insieme logico parzialmente definito {1,a,{}|R}
// con nome S
LSet set4 = new LSet().ins(1).ins(a).ins(LSet.empty()).setName("S");
```

Gli oggetti di tipo `LSet` supportano un gran numero di vincoli, quali quelli di uguaglianza (`eq`), disuguaglianza (`neq`), appartenenza (`in`), non appartenenza (`nin`). disgiunzione (`disj`), non disgiunzione (`ndisj`) e unione (`union`); inoltre supportano anche alcuni vincoli derivati, ovvero definiti in funzione dei precedenti, quali `diff, inters, less, size, subset`.

4.1.3 La classe `Constraint`

All'interno della libreria di JSetL, i vincoli diventano istanze della classe `Constraint`, un contenitore di oggetti di tipo `AConstraint` (Atomic Constraint) i quali rappresentano un singolo vincolo atomico. Un `Constraint`, di conseguenza, è formato da almeno un `AConstraint` e ci permette di tenere in memoria un dato insieme di vincoli.

Di seguito si mostrano alcuni esempi relativi alla creazione di un `LSet`:

```
// Creazione di un vincolo senza nome
Constraint c1 = new Contraint(set3.eq(set1));
// Creazione di un vincolo di nome "vincolo"
Constraint c2 = new Contraint("vincolo",set3.eq(set1));
// Creazione di un vincolo composto
Constraint c3 = new Contraint(set3.eq(set1).and(set2.eq(LSet.empty())));
```

4.1.4 La classe `LPair`

La classe `LPair` permette di realizzare una coppia logica (x, y) dove x e y sono oggetti qualsiasi. `LPair` è implementata come estensione della classe `LList`, la quale permette l'implementazione di liste logiche in grado di contenere qualsiasi oggetto. Nella costruzione di oggetti di tipo `LList` è importante l'ordine degli elementi: liste con gli stessi valori differiscono se essi sono messi in ordine diverso $[1, a, ...] \neq [a, 1, ...]$ e di conseguenza questo vale anche per gli elementi di LPair. Si possono creare LPair bound o unbound, con o senza

un nome esterno identificativo.
`LPair` eredita i vincoli di uguaglianza e disuguaglianza di `LList` e implementa i vincoli di appartenenza e non appartenenza.
Seguono alcuni esempi riguardanti la creazione di `LPair`:

```
// Creazione di una coppia logica non inizializzata e con nome P1
LPair p1 = new LPair("P1");
// Creazione di una coppia logica inizializzata senza nome
LPair p2 = new LPair(1, new LVar());
// Creazione di una coppia logica inizializzata con nome P2
LPair p3 = new LPair("P2", new LVar(1), 5);
// Creazione di una coppia logica inizializzata con un altro LPair
LPair p4 = new LPair(p2);
// Creazione di una coppia logica bound con un altro LPair con nome P5
LPair pp5 = new LPair("P5",pp3);
```

N.B. Attraverso il costruttore `LPair(LPair p)` è possibile creare una nuova coppia logica uguale a `p` similmente al funzionamento del vincolo `eq()`.

Data una coppia ordinata di valori (x, y) è possibile accedere ai singoli valori attraverso i seguenti metodi:

- `Object getFirst()` ritorna il primo elemento della coppia;
- `Object getSecond()` ritorna il secondo elemento della coppia.

4.1.5 La classe del risolutore: `Solver`

La classe `Solver` implementa il risolutore di vincoli per il linguaggio. Attraverso di essa, è possibile aggiungere vincoli, controllarne la soddisfacibilità ed eventualmente controllare quali vincoli sono presenti nel "constraint store" in un dato momento. I vincoli vengono risolti attraverso le regole di riscrittura. La procedura di risoluzione termina quando tutti i vincoli sono in forma irriducibile (non ci sono più regole di riscrittura da applicare), oppure se viene lanciata un'eccezione di tipo `Failure`, che accade nel caso in cui il vincolo

non sia soddisfacibile.
Alcuni attributi protected presenti nella classe:

- `storeInventario`, un campo booleano utilizzato per indicare se lo store del solver è stato modificato;
- `storeSize`, un campo utilizzato per controllare la dimensione attuale dello store del risolutore, ovvero quanti vincoli contiene;
- `protected ConstraintStore store = new ConstraintStore(this)` rappresenta lo store del risolutore;

Alcuni metodi public presenti nella classe:

- `void add(Constraint c)`: aggiunge il vincolo `c` al `ConstraintStore` del risolutore;
- `void showStore()`: stampa un insieme di tutti i vincoli presenti nel `ConstraintStore` che ancora non sono stati risolti;
- `boolean check(Constraint c)`: controlla che i vincoli nel `ConstraintStore` del risolutore e il vincolo `c` da aggiungere siano soddisfacibili. Se ciò risulta possibile, viene calcolata una soluzione e aggiunto il vincolo al risolutore e il metodo ritorna `true`, altrimenti il vincolo è insoddisfacibile, e i vincoli nel risolutore rimangono invariati, il vincolo `c` viene scartato ed il metodo restituisce false come suo risultato;
- `void solve()`: risolve i vincoli nel `ConstraintStore` del risolutore. Come per il metodo `check()` si cerca una possibile soluzione, se essa non viene trovata il metodo lancia un'eccezione di tipo `Failure`.

4.1.6 Il funzionamento di JSetL

Si riporta ora un semplice esempio di programma realizzato in Java che sfrutta la programmazione a vincoli della libreria JSetL.
Il programma crea un insieme chiamato `set` che contiene alcuni oggetti ed

un resto `_rest` il cui contenuto è sconosciuto. Il programma crea anche un `solver` a cui verrà aggiunto il vincolo `x` $\in$ `set` per eventualmente assegnare un nuovo valore all'insieme. Viene poi stampato il contenuto (modificato) dell'insieme `set` e lo store del solver tramite i rispettivi metodi `set.output()` e `solver.showStore()`.

```
import JSetL.*;

public static void main(String[] args) throws Failure{
   Solver solver = new Solver();
   LVar x = new LVar(3).setName("X"); // X = 3

   LSet set = new LSet("rest")  // set = {{15}, c, hello| _rest}
      .ins("hello")
      .ins('c')
      .ins(LSet.empty().ins(15)).setName("set");

   //aggiungo un nuovo vincolo al solver
   solver.add(x.in(set));

   // mostro i vincoli presenti nello store del solver
   solver.showStore();

   //risolvo la congiunzione di vincoli presenti nello store del solver
   solver.solve();

   //stampo il contenuto di set dopo la risoluzione dei vincoli
   set.output();

   /*
       OUTPUT PROGRAMMA
       Store: 3 in {{15}, c, hello|_rest}
       set = {{15}, c, hello, 3|_?}
   */
}
```

4.2 Il Prodotto Cartesiano: La nuova classe `CP`

La classe `CP` implementa il nuovo termine del linguaggio: il prodotto Cartesiano. Un prodotto Cartesiano è un particolare oggetto logico realizzato estendendo la classe delle relazioni logiche `LRel`, che a sua volta estende la classe degli insiemi logici `LSet`, ed è composto da due oggetti di tipo `LSet` che rappresentano insiemi di elementi di tipo arbitrario.
Un `CP` è costituito idealmente da un insieme di coppie (ovvero, una relazione binaria), formate da una combinazione dei valori degli insiemi che lo formano. Formalmente, il prodotto cartesiano $A \times B$ rappresenta l'insieme $\{(x, y) : x \in A \wedge y \in B\}$.
Di conseguenza se anche solo uno dei due insiemi è vuoto, l'intero prodotto `CP` risulta essere vuoto. Si noti che gli elementi all'interno degli `LSet` possono generare coppie uguali ripetute: per esempio un prodotto tra due insiemi $\{1, 2\}$ e $\{2, 2\}$ genera l'insieme di coppie $\{(1, 2), (1, 2), (2, 2), (2, 2)\}$ e l'ordine di tali coppie è arbitrario e non viene tenuto in considerazione durante le operazioni sulla classe.
Non sempre è possibile e/o opportuno espandere un `CP` nell'insieme di coppie corrispondente. Ad es. se uno dei due insiemi componenti un `CP` contiene delle variabili non inizializzate o è esso stesso una variabile non inizializzata, allora l'espansione non è proprio possibile. Dunque, i due insiemi componenti un `CP` sono memorizzati separatamente, e viene fatta un'espansione del `CP` solo se strettamente necessario, tramite l'opportuna funzione `expand()`.
I metodi forniti dalla classe `CP` sono sostanzialmente gli stessi di quelli della classe `LRel`, alcuni ereditati, e altri sovrascritti per riadattarli opportunamente. In particolare, sono forniti costruttori, metodi di output e di utilità.

4.2.1 Costruttori

La classe fornisce tre tipi di costruttori:

- Costruttore senza parametri.
- Costruttore con due `LSet` come parametro.
- Costruttore con un `CP` come parametro (costruttore di copia).

Un costruttore senza parametri si utilizza nel seguente modo:

```
// Utilizzo del costruttore CP()
CP cp1 = new CP();
// Utilizzo del costruttore CP(String n)
CP cp2 = new CP(name);
```

Esso restituisce un `CP` unbound, con o senza un nome esterno:

- Il parametro `name` è opzionale, e viene utilizzato per dare un nome esplicito al `CP`, se omesso viene dato un nome in automatico dalla libreria. Il parametro `name` può essere solamente di tipo `string`.

Un costruttore con due insiemi `LSet` si utilizza nel seguente modo:

```
// Utilizzo del costruttore CP(LSet a, LSet b)
CP cp1 = new CP(setA, setB);
// Utilizzo del costruttore CP(String n, LSet a, LSet b)
CP cp2 = new CP(name, setA, setB);
```

Esso restituisce un `CP` bound, con o senza un nome di riferimento, i cui due insiemi componenti sono inizializzati dall'utente in modo esplicito:

- il parametro `name` funziona come definito prima;
- il parametro `setA` rappresenta il primo insieme che compone il `CP`. Deve essere un oggetto di classe `LSet` o sua sottoclasse;
- il parametro `setB` rappresenta il secondo insieme che compone il `CP` e deve essere un oggetto di classe `LSet` o sua sottoclasse.

Un costruttore con un oggetto `CP` si utilizza nel seguente modo:

```
// Utilizzo del costruttore CP(CP cp)
CP cp1 = new CP(cp);
// Utilizzo del costruttore CP(String n, CP cp)
CP cp2 = new CP(name, cp);
```

Esso restituisce un nuovo `CP`, con o senza un nome di riferimento, con lo stesso valore dell'insieme passato come parametro; non viene effettuata una vera copia, ma il nuovo `CP` viene collegato ai valori del parametro tramite un campo `equ`:

- il parametro `name` funziona come definito prima.
- il parametro `cp` rappresenta un oggetto di classe `CP` di cui vogliamo "prendere" i valori ed utilizzarli per un nuovo prodotto cartesiano. Questo ci permette di simulare il funzionamento di un costruttore di copia, creando un legame tra il cp passato come parametro e il cp nuovo, creato senza effettivamente andare a copiare i valori degli insiemi.

4.2.2 Metodi di supporto

Oltre ai metodi di creazione, gli oggetti di classe `CP` sono dotati di metodi di supporto che gli permettono di svolgere diverse operazioni. Alcuni di questi metodi, come per altri già citati, sono disponibili anche per altre classi della libreria, e vengono qui estesi e resi compatibili anche con i prodotti cartesiani.

I seguenti metodi sono gli stessi presenti anche nelle classi `LSet/LRel` opportunamente estesi:

- `CP clone()`, restituisce un nuovo CP identico a quello su cui la funzione è chiamata
- `boolean equals(CP cp)`, ritorna `true` se questo `CP` e `cp` sono equivalenti, `false` altrimenti.
  N.B. L'ordine delle coppie è ininfluente.

- `LSet getFirstSet()`, ritorna il valore della prima componente di questo `CP`.
- `LSet getSecondSet()`, ritorna il valore della seconda compomente di questo `CP`.
- `LSet getName()`, ritorna il nome esterno di questo `CP`.
- `boolean isBound()`, ritorna `true` se questo `CP` è legato a dei valori, `false` altrimenti.
- `void output()`, stampa a schermo questo `CP`.
- `CP setName(String extName)`, assegna `extName` al nome esterno di questo `CP` e restituisce questo `CP` modificato.
- `String toString()`, ritorna una stringa corrispondente a questo `CP`;

La classe `CP` fornisce tutti i metodi di gestione della collezione di dati delle classi `LSet/LRel`. Questi metodi vengono utilizzati nella gestione e implementazione dei vincoli logici di cui parleremo nel capitolo successivo.

- `int getSize()`, restituisce un intero che rappresenta il numero di tutte le coppie generate dal prodotto cartesiano rappresentato da questo `CP`.
- `boolean isEmpty()`, restituisce `true` se questo `CP` è inizializzato (bound) e vuoto, `false` altrimenti.
- `boolean isGround()`, restituisce `true` se questo `CP` è formato da insiemi completamente specificati, ovvero i cui elementi hanno tutti un valore definito e non variabile, `false` altrimenti.
- `LSet expand()`, restituisce l'insieme `LSet` corrispondente al `CP` costituito dalle coppie di elementi calcolati esplicitamente mediante il prodotto cartesiano. L'operazione può risultare costosa e può essere realizzata solamente su oggetti `CP` completamente specificati (ground), altrimenti il metodo lancia l'eccezione `NotInitVarException`.

4.2.3 Elaborazione di un `CP`

Per operare su un oggetto di classe `CP` si utilizzano gli stessi metodi disponibili per gli oggetti di tipo `LSet/LRel`, eventualmente riadattati, sfruttando l'ereditarietà tra le due classi

E' importante notare che la classe `CP` supporta solamente operazioni su coppie di elementi, di conseguenza, aggiungendo una coppia di valori ad un `CP` mediante un metodo `ins(LPair p )`, non avremo più un prodotto cartesiano tra due insiemi, ma un insieme formato da coppie.

Notare che poichè l'ordine delle coppie e degli elementi memorizzati nei due `LSet` non è importante, non è necessario definire metodi distinti per inserimenti in testa o in coda, poichè produrrebbero lo stesso risultato finale.

Esempio di uso 1

```
LSet a = LSet.empty().ins(2).ins(1);        // {2,1}
LSet b = LSet.empty().ins(3);             // {3}
CP cp = new CP("cp",a,b);               // cp = {1,2} x {3}
LSet S = cp.ins(new LPair(1,4)).setName("S"); // {[1,4]|{1,2} x {3}}
```

Esempio di uso 2

```
LSet A = LSet.empty().ins(1).ins(2);   // {1,2}
LSet B = LSet.empty().ins('a').ins('b'); // {a,b}
CP cp1 = new CP("cp1",A,B);             // cp1 = {2,1} x {b,a}
LSet D = new LSet("D");              // insieme D unbound
CP cp2 = new CP("cp2",A,D);             // {2,1} x D
LSet E = cp2.ins(new LPair(5,6)).setName("E"); // {[5,6]|{2,1} x D}
```

Esempio di uso 3

```
LSet a = LSet.empty().ins(2).ins(1);
LSet b = LSet.empty().ins(3);
CP cp1 = new CP("cp",a,b);          // cp = {1,2} x {3}
CP cp2 = new CP("cp_new", cp1);       // cp_new = {1,2} x {3}
```

Capitolo 5

Trattamento di vincoli su CP in JSetL

La libreria JSetL dispone di diverse funzionalità per definire vincoli tra gli oggetti logici, che verranno in risolti tramite la classe `Solver`.
La risoluzione dei vincoli all'interno della libreria avviene applicando opportune regole di riscrittura, le stesse riportate nel capitolo 3. In particolare per la classe `CP` sono state implementate regole di riscrittura per vincoli di base quali uguaglianza, disuguaglianza, appartenenza, non appartenenza, disgiunzione, e unione. La classe `CP` supporta anche vincoli definiti in funzione dei vincoli base, quali non unione, non disgiunzione, non sottoinsieme, non intersezione e non differenza, i quali godono di grande generalità, in quanto compatibili con tutti le tipologie di insiemi estensionali presenti nella libreria.

5.1 Il Vincolo di Uguaglianza

Il vincolo di uguaglianza ha la forma `eq(A,B)`, dove A e B sono insiemi estensionali e almeno uno dei due è di classe `CP`. Il vincolo risulta soddisfatto se e solo se i due insiemi A e B risultano avere gli stessi elementi nel loro insieme.

Il caso in cui solamente uno dei due argomenti sia un oggetto `CP`, viene gestito in via generale dalla seguente regola già vista nel capitolo 3:

$$A \times B = \{z_1 \sqcup Z_1\} \rightarrow A = \{n_1 \sqcup N_1\} \wedge B = \{n_2 \sqcup N_2\} \wedge z = (n_1, n_2) \wedge un(\{n_1\} \times N_2, N_1 \times \{\{n_1\} \sqcup N_2\}, Z)$$

o dalla sua simmetrica

$$\{z_1 \sqcup Z_1\} = A \times B \rightarrow A = \{n_1 \sqcup N_1\} \wedge B = \{n_2 \sqcup N_2\} \wedge z = (n_1, n_2) \wedge un(\{n_1\} \times N_2, N_1 \times \{\{n_1\} \sqcup N_2\}, Z)$$

Se, invece, entrambi gli elementi passati come parametro nel vincolo sono degli oggetti `CP`, si fa riferimento alla regola

$$A \times B = C \times D \rightarrow A = C \wedge B = D$$

5.1.1 Uguaglianza in JSetL

In JSetL, le regole per l'uguaglianza sono implementate all'interno della classe `RwRulesEq`, tramite due metodi: uno per gestire il caso di un oggetto `CP` messo a confronto con un altro insieme differente della libreria, e uno per gestire il caso di due oggetti `CP`. Il primo metodo è implementato nel seguente modo.

Un primo `if` gestisce il caso base dove l'insieme `Set` passato come parametro è vuoto.

```
protected void eqCP(CP cp, LSet set, AConstraint s) throws Failure {
   ...
   // X x Y = {}
   if (set.isInit() && set.isEmpty()) {
      switch(s.caseControl) {
         case 0:    // --- > X = {}
            Solver.addChoicePoint(s);
            s.arg1 = cp.getFirstSet();
            s.cons = Environment.eqCode;
            s.arg2 = LSet.empty();
            s.caseControl = 0;
```

```
            Solver.storeInvariato = false;
            return;
         case 1:    // ---> Y = {}
            s.arg1 = cp.getSecondSet();
            s.cons = Environment.eqCode;
            s.arg2 = LSet.empty();
            s.caseControl = 0;
            Solver.storeInvariato = false;
            return;
      }
```

Il secondo `if` gestisce il caso in cui l'oggetto `CP` è vuoto.

```
   }
   // X x {} = Z ---> Z = {}
   // {} x Y = Z ---> Z = {}
   else if (cp.isInit() && cp.isEmpty()) {
      s.arg1 = set;
      s.cons = Environment.eqCode;
      s.arg2 = LSet.empty();
      Solver.storeInvariato = false;
      return;
   }
```

Il terzo `if` gestisce il caso in cui si ha un insieme inizializzato ma non vuoto vincolato ad un `CP` variabile.

```
   //X x Y = {z u Z} --->
   if (set.isInit() && !set.isEmpty()){
      Object o = set.getOne();
      LVar n1 = new LVar();
      LVar n2 = new LVar();
      LPair z = new LPair(n1,n2);
      s.arg1 = o;
      s.cons = Environment.eqCode;        // z = (n1,n2)
      s.arg2 = z;

      LSet N1 = new LSet();
```

```
      Solver.add(new AConstraint(cp.getFirstSet(), Environment.eqCode, N1.
          ins(n1)));
      // X = {n1 u N1}
      LSet N2 = new LSet();
      Solver.add(new AConstraint(cp.getSecondSet(), Environment.eqCode, N2
          .ins(n2)));
      // Y = {n2 u N2}

      Solver.add(new Constraint(new AConstraint(
         new CP(LSet.empty().ins(n1), N2),
         Environment.unionCode,
         new CP(N1, N2.ins(n2)),
         set.removeOne()))); //un({n1} x N2, N1 x {n2 u N2}, Z}

      Solver.storeInvariato = false;
      return;
   }
   ...
   return;
}
```

Il secondo metodo, chiamato se ci si ritrova a dover confrontare due `CP`, è definito nel seguente modo:

```
protected void eqCP(CP cp1, CP cp2, AConstraint s) throws Failure {
   ...
   //W x X = Y x Z ---> W = Y && X = Z
   if(cp1.isInit() || cp2.isInit()) {
      s.arg1 = cp1.getFirstSet();
      s.cons = Environment.eqCode;
      s.arg2 = cp2.getFirstSet();
      Solver.add(new AConstraint(cp1.getSecondSet(), Environment.eqCode,

         cp2.getSecondSet()));
      Solver.storeInvariato = false;
      return;
      }
```

```
   ...
}
```

La decisione di quale metodo invocare per il vincolo viene stabilita tramite la funzione `eq(s)` dove `s` è un oggetto di classe `AConstraint`, il quale racchiude un vincolo generale, il quale, una volta analizzato tramite dei costrutti `if` a cascata, ci permette di capire in particolare quali classi devono essere messe a confronto.
In particolare:

```
protected void eq(AConstraint s) throws Failure {
   ...
   else if (s.arg1 instanceof CP && s.arg2 instanceof CP)
      eqCP((CP) s.arg1 , (CP) s.arg2, s);      // cp = cp
   else if (s.arg1 instanceof CP && s.arg2 instanceof LSet)
      eqCP((CP) s.arg1 , (LSet) s.arg2, s);    // cp = lset
   else if (s.arg1 instanceof LSet && s.arg2 instanceof CP) {
      Object tmp = s.arg1;                     // lset = cp
      s.arg1 = s.arg2;
      s.arg2 = tmp;
      eq(s);
   }
   ...
   return;
}
```

5.1.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo di uguaglianza in qualche programma d'esempio. In essi si suppone di aver costruito un oggetto risolutore `Solver` tramite lo statement `Solver solver = new Solver();`

Esempio 1

In questo esempio, un `CP` viene confrontato con un `LSet`

```
LSet a = LSet.empty().ins(3).ins(1);
LSet b = LSet.empty().ins(2).ins(4);
CP cp = new CP("cp",a,b);        // cp = {3,1} x {2,4}

LSet R = new LSet("R");
LVar X = new LVar("X");
LSet S = R.ins(X).setName("S");    // S = {_X|_R}

solver.add(S.eq(cp));
Solver.solve();               // S = {[1,4],[1,2],[3,4],[3,2]}
```

Esempio 2

In questo esempio, un `CP` viene confrontato con un altro `CP`

```
LSet A = LSet.empty().ins(1);
LSet B = LSet.empty().ins(2);
CP c1 = new CP(A,B);    // c1 = {1,2} x {3,4}

LSet C = new LSet();
LSet D = new LSet();
CP c2 = new CP(C,D);    // c2 = {}

solver.solve(c1.eq(c2));   // c2 = {1,2} x {3,4}
```

Esempio 3

In questo esempio, un `CP` viene confrontato con un `LSet` contenente una variabile. Il programma stamperà tutte le soluzioni possibili del vincolo, ovvero un `CP` privato di una coppia, la quale viene stampata separatamente e messa nella variabile `X`

```
LSet a = LSet.empty().ins(3).ins(1);
LSet b = LSet.empty().ins(2).ins(4);
CP cp = new CP("cp",a,b);
cp.output();          // {1,3} x {4,2}
cp.expand().output();    // {[1,4],[1,2],[3,4],[3,2]}
```

```
LSet R = new LSet("R");
LVar X = new LVar("X");
LSet S = R.ins(X).setName("S");
S.output();           // S = {_X|_R}

solver.add(S.eq(cp));    // {_X|_R} = {1,3} x {4,2}
solver.add(X.nin(R));    // X nin R

if (solver.check())
System.out.println("true");
else
System.out.println("false");
X.output();           // X = [1,4]
R.output();           // R = {[1,2],[3,4],[3,2]}

solver.nextSolution();
X.output();           // X = [1,2]
R.output();           // R = {[1,4],[3,2],[3,4]}

solver.nextSolution();
X.output();           // X = [3,4]
R.output();           // R = {[3,2],[1,4],[1,2]}
```

5.2 Il Vincolo di Disuguaglianza

Il vincolo di disuguaglianza ha la forma `neq(A,B)`, dove A e B sono insiemi estensionali e almeno uno dei due è di classe `CP`. Il vincolo risulta soddisfatto se e solo se i due insiemi A e B risultano avere almeno un elemento differente tra loro.

In Java, le regole per la gestione degli oggetti di classe `CP` vengono implementate tramite un unico metodo che sfrutta l'ereditarietà della classe `CP` dalla classe `LSet`, occupandosi principalmente di rispettare la seguente regola di riscrittura:

$$A \times B \neq Z \rightarrow (n \in A \times B \wedge n \notin Z) \vee (n \notin A \times B \wedge n \in Z)$$

5.2.1 Disuguaglianza in JSetL

Come per l'uguaglianza, le regole per la disuguaglianza sono implementate nella classe `RwRulesEq` come segue:

```
private void neqCP(CP cp, LSet set, AConstraint s) throws Failure{
  ...
  if(set.isInit() && cp.isInit()) {
     Object n = set.getOne();
     if(n instanceof LPair) {
        switch(s.caseControl) {
           case 0:
              Solver.addChoicePoint(s);
              s.arg1 = n;
              s.cons = Environment.inCode;
              s.arg2 = cp;
              Solver.add(new AConstraint(n, Environment.ninCode, set));
              Solver.storeInvariato = false;
              break;
           case 1:
              s.caseControl = 0;
              s.arg1 = n;
              s.cons = Environment.ninCode;
              s.arg2 = cp;
              Solver.add(new AConstraint(n, Environment.inCode, set));
              Solver.storeInvariato = false;
              break;
        }
        return;
     }
  }
  ...
}
```

Da notare che, a differenza dell'uguaglianza, ci basta un'unica funzione per poter descrivere il vincolo, non sono necessarie ulteriori distinzioni tra diverse casistiche.

La decisione di quale metodo invocare per il vincolo viene stabilita tramite la funzione `neq(s)`, dove `s` è un oggetto di classe `AConstraint`, che ci permette di capire quali classi devono essere messe a confronto.
In particolare:

```
protected void neq(AConstraint s) throws Failure {
   ...
   else if (s.arg1 instanceof CP && s.arg2 instanceof LSet)
      neqCP((CP) s.arg1, (LSet)s.arg2, s);
   else if(s.arg1 instanceof LSet && s.arg2 instanceof CP) {
      Object tmp = s.arg1;
      s.arg1 = s.arg2;
      s.arg2 = tmp;
      neqCP((CP) s.arg1, (LSet) s.arg2, s);
   ...
}
```

5.2.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo di disuguaglianza in qualche programma d'esempio. Si suppone sempre di avere un oggetto `Solver` già istanziato.

Esempio 1
In questo esempio, un `CP` inizializzato viene confrontato con un `LSet` inizializzato, dove il vincolo è rispettato.

```
   Solver solver = new Solver();

   LSet X = LSet.empty().ins(2);
   LSet Y = LSet.empty().ins(3);
   CP c = new CP(X,Y);                    // {2} x {3}
   LSet Z = LSet.empty().ins(new LPair(2,'a')); // {(2,'a)}
```

```
solver.solve(c.neq(Z));  // true, gli insiemi hanno elementi differenti
```

Esempio 2

In questo esempio, un `CP` inizializzato viene confrontato con un `LSet` inizializzato, dove il vincolo non è rispettato.

```
Solver solver = new Solver();

LSet X = LSet.empty().ins(2);
LSet Y = LSet.empty().ins('a');
CP c = new CP(X,Y);                    // {2} x {'a'}
LSet Z = LSet.empty().ins(new LPair(2,'a')); // {(2,'a)}

solver.solve(c.neq(Z)); // false, gli insiemi hanno gli stessi elementi
```

5.3 Il Vincolo di Appartenenza

Il vincolo di appartenenza ha la forma `in(t,A)`, dove A è un insieme di classe `CP` qualsiasi e t indica un oggetto che può essere o una variabile `LVar`, o una coppia di valori `LPair`. Il vincolo risulta soddisfatto se e solo se t appartiene all'oggetto `CP` in questione.
In JSetL, le regole per la gestione degli oggetti di classe `CP` sono suddivise in due metodi principali, che gestiscono separatamente il caso in cui t sia un oggetto `LPair` da quello dove t sia una variabile `LVar`.

Il vincolo di appartenenza segue una regola di riscrittura generale, definita come

$$z \in A \times B \rightarrow z = (n_1, n_2) \wedge n_1 \in A \wedge n_2 \in B$$

5.3.1 Appartenenza in JSetL

Il vincolo di appartenenza in JSetL è implementato all'interno della classe `RwRulesSet` e suddiviso in due metodi per gestire separatamente il caso in cui il termine t sia variabile dal caso in cui sia una coppia di valori (anche variabili).
Per il parametro di tipo `LPair` si utilizza la funzione:

```
private void inLPairCP(LPair z, CP cp, AConstraint s) throws Failure {
  ...
  if(!cp.isInit() || cp.isEmpty() )
     Solver.fail(s);
     return;
  }
  else if(cp.isInit()) {
     LVar n1 = new LVar();
     LVar n2 = new LVar();
     LPair z1 = new LPair(n1,n2);
     s.arg1 = z1;
     s.cons = Environment.eqCode;        // z = (n1,n2)
     s.arg2 = z;
     Solver.add(new AConstraint(n1, Environment.inCode, cp.getFirstSet())
         );
     Solver.add(new AConstraint(n2, Environment.inCode, cp.getSecondSet()
         ));
  }
  else {
     Solver.fail(s);    // fallimento del vincolo
     return;
  }
  ...
}
```

Mentre per gestire il caso con un parametro di tipo `LVar` si utilizza la funzione:

```
private void inLVarCP(LVar v, CP cp, AConstraint s) throws Failure {
  ...
  LVar n1 = new LVar();
```

```
   LVar n2 = new LVar();
   LPair z = new LPair(n1,n2);
   s.arg1 = v;
   s.cons = Environment.eqCode;        // z = (n1,n2)
   s.arg2 = z;
   Solver.add(new AConstraint(z, Environment.inCode, cp));
   Solver.storeInvariato = false;
   ...
 }
```

La decisione di quale metodo invocare per il vincolo viene stabilita tramite la funzione `in(s)` dove `s` è un oggetto di classe `AConstraint`, che ci permette di capire in particolare quali classi devono essere messe a confronto. In particolare:

```
protected void in(AConstraint s) throws Failure {
   ...
   else if (s.arg1 instanceof LVar && s.arg2 instanceof CP)
      inLVarCP((LVar)s.arg1, (CP)s.arg2, s);
   else if (s.arg1 instanceof LPair && s.arg2 instanceof CP)
      inLPairCP((LPair)s.arg1, (CP)s.arg2, s);
   ...
   return;
}
```

5.3.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo di appartenenza in qualche programma d'esempio.

Esempio 1

In questo esempio, si mostra un semplice funzionamento del vincolo per `LPair` dove si controlla che una coppia appartenga al prodotto.

```
   Solver solver = new Solver();
   LSet A = LSet.empty().ins(1).ins(2);
   LSet B = LSet.empty().ins(3).ins(4);
```

```
CP cp = new CP(A,B);      // cp = {1,2} x {3,4}

LPair x = new LPair(1,3); // x = [1,3]
x.setName("x");
cp.setName("cp");

solver.solve(x.in(cp));  // true
```

Esempio 2

In questo esempio, si mostra un semplice funzionamento del vincolo per `LVar` dove si controlla che una variabile appartenga al prodotto.

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(1).ins(2);
LSet B = LSet.empty().ins(3).ins(4);
CP cp = new CP(A,B);      // cp = {1,2} x {3,4}

LPair Y = new LPair(1,3);
LVar Y1 = new LVar("Y1", Y);

solver.solve(Y1.in(cp)); // true
```

Esempio 3

In questo esempio, abbiamo un `LVar` viene vincolato ad essere un valore del `CP` inizializzato.

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(1).ins(2);
LSet B = LSet.empty().ins(3).ins(4);
CP cp = new CP(A,B);  // cp = {1,2} x {3,4}

LVar Z = new LVar("Z");
solver.solve(Z.in(cp)); // Z in cp

Z.output();  //   Z = [2,4]
solver.nextSolution();
Z.output();  // Z = [2,3]
solver.nextSolution();
Z.output();  // Z = [1,4]
```

```
solver.nextSolution();
Z.output();   // Z = [1,3]
```

5.4 Il Vincolo di Non Appartenenza

Il vincolo di non appartenenza ha la forma `nin(t,A)`, dove `A` è un insieme di classe `CP` qualsiasi e t indica un oggetto che può essere o una variabile `LVar`, o una coppia di valori `LPair`. Il vincolo risulta soddisfatto se e solo se t non appartiene all'oggetto `CP`.
In JSetL, le regole per il trattamento del vincolo di non appartenenza su oggetti di classe `CP` sono suddivise in due metodi principali, come per il suo positivo `in`, che gestiscono separatamente il caso in cui termine `t` sia un oggetto `LPair` o una variabile `LVar`.

Il vincolo di non appartenenza segue la regola di riscrittura generale definita come

$$z \notin A \times B \rightarrow z = (n_1, n_2) \wedge (n_1 \notin A \vee n_2 \notin B)$$

5.4.1 Non Appartenenza in JSetL

Si riporta ora nel dettaglio l'implementazione del vincolo per entrambi i casi all'interno della classe `RwRulesSet`.
Per `LPair`:

```
private void ninLPairCP(LPair z, CP cp, AConstraint s) throws Failure {
   ...
   if(cp.isInit() && cp.isEmpty()) {
      s.solved = true;
      return;
   }
   else if(cp.isInit() && !cp.isEmpty() && z.isInit()) {
      Object n1 = z.getFirst();
      Object n2 = z.getSecond();
```

```
      switch(s.caseControl) {
         case 0:
            Solver.addChoicePoint(s);
            s.arg1 = n1;
            s.cons = Environment.ninCode;
            s.arg2 = cp.getFirstSet();
            Solver.storeInvariato = false;
            return;
         case 1:
            s.caseControl = 0;
            s.arg1 = n2;
            s.cons = Environment.ninCode;
            s.arg2 = cp.getSecondSet();
            Solver.storeInvariato = false;
            return;
      }
   }
   else {
      Solver.fail(s);
      return;
   }
   ...
}
```

Per `LVar`:

```
private void ninLVarCP(LVar v, CP cp, AConstraint s) throws Failure {
   ...
   LVar n1 = new LVar();
   LVar n2 = new LVar();
   LPair z = new LPair(n1,n2);
   s.arg1 = v;
   s.cons = Environment.eqCode;        // z = (n1,n2)
   s.arg2 = z;
   Solver.add(new AConstraint(z, Environment.ninCode, cp));
   Solver.storeInvariato = false;
   ...
}
```

La decisione di quale metodo invocare per il vincolo viene stabilita tramite la funzione `nin(s)` dove `s` è un oggetto di classe `AConstraint`, che ci permette di capire quali classi devono essere messe a confronto.
In particolare:

```
protected void nin(AConstraint s) throws Failure {
   ...
   else if (s.arg1 instanceof LVar && s.arg2 instanceof CP)
      ninLVarCP((LVar)s.arg1, (CP)s.arg2, s);
   else if (s.arg1 instanceof LPair && s.arg2 instanceof CP)
      ninLPairCP((LPair)s.arg1, (CP)s.arg2, s);
   ...
   return;
}
```

5.4.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo di non appartenenza in qualche programma d'esempio.

Esempio 1

In questo esempio, si mostra un semplice funzionamento del vincolo per `LVar` dove si pone la non appartenenza di una variabile z in un prodotto cartesiano

```
   Solver solver = new Solver();
   LSet A = LSet.empty().ins(1).ins(2);
   LSet B = LSet.empty().ins(3).ins(4);
   CP cp = new CP(A,B);                // {2,1} x {4,3}
   LVar z = new LVar(new LPair(1,7));  // z = [1,7]
   solver.solve(z.nin(cp));
```

Esempio 2

In questo esempio, si mostra un semplice funzionamento del vincolo per `LPair` dove si pone la non appartenenza di una coppia z in un prodotto cartesiano

```
   Solver solver = new Solver();
   LSet A = LSet.empty().ins(1).ins(2);
```

```
LSet B = LSet.empty().ins(3).ins(4);
CP cp = new CP(A,B);
LPair z = new LPair(2,1);
solver.solve(z.nin(cp));
```

5.5 Il Vincolo di Disgiunzione

Il vincolo di disgiunzione ha la forma `disj(A,B)`, dove A e B sono insiemi qualsiasi, e almeno uno dei due è di classe `CP`. Il vincolo risulta soddisfatto se e solo se gli insiemi A e B risultano disgiunti, ovvero se la loro intersezione è equivalente all'insieme vuoto $\emptyset$.
In JSetL, le regole per il trattamento del vincolo di non appartenenza su oggetti `CP` sono implementate sfruttando l'ereditarietà della classe `CP` dagli insiemi di tipo `LSet`, ovvero implementate seguendo, principalmente, le regole:

$$\text{If } Z \not\equiv \emptyset : \{a \sqcup A\} \times \{b \sqcup B\} \parallel Z \rightarrow \{(a,b) \sqcup N\} \parallel Z \wedge un(a \times B, A \times \{b \sqcup B\}, N)$$

o dalla sua inversa

$$\text{If } Z \not\equiv \emptyset : Z \parallel \{a \sqcup A\} \times \{b \sqcup B\} \rightarrow \{(a,b) \sqcup N\} \parallel Z \wedge un(a \times B, A \times \{b \sqcup B\}, N)$$

5.5.1 Disgiunzione in JSetL

Le regole vengono implementate nella libreria all'interno della classe `RwRulesSet` attraverso il seguente singolo metodo:

```
private void disjCP(CP cp, LSet set, AConstraint s) throws Failure {
   ...
   // Tutti gli argomenti variabili
   if(cp.isVariable() && set.isVariable()) {
      s.solved = true;
```

```
    return;
  }
  // Argomenti inizializzati ma con il set vuoto
  else if(cp.isInit() && set.isInit() && set.isEmpty()) {
    s.solved = true;
    return;
  }
  // Argomenti inizializzati ma con il CP vuoto
  else if(cp.isInit() && cp.isEmpty() && set.isInit()) {
    s.solved = true;
    return;
  }
  // Caso principale
  else if(cp instanceof CP && set instanceof LSet)
    if (!set.isInit() || set instanceof CP) {
      Object x = cp.getFirstSet().getOne();
      Object y = cp.getSecondSet().getOne();
      LPair p = new LPair(x,y);
      LSet N = LSet.empty().ins(p);

      s.arg1 = N;
      s.cons = Environment.disjCode;
      s.arg2 = set;

      Solver.add(new Constraint(new AConstraint(
      new CP(LSet.empty().ins(x), cp.getSecondSet()),
        Environment.unionCode,
        new CP(cp.getFirstSet(), cp.getSecondSet().ins(y)),
        N)));
      Solver.storeInvariato = false;
      return;
    }
    else {
      LPair z = (LPair)set.getOne();
      set = set.getResto();
      s.arg1 = z;
      s.cons = Environment.ninCode;
      s.arg2 = cp;
```

```
        Solver.add(new AConstraint(cp, Environment.disjCode, set));
        Solver.storeInvariato = false;
        return;
      }
      return;
   }
   ...
}
```

L'invocazione del metodo che si occupa del vincolo di disgiunzione viene stabilita tramite la funzione `disj(s)`, con `s` di classe `AConstraint`, che ci permette di capire quali classi devono essere messe a confronto.
In particolare:

```
protected void disj(AConstraint s) throws Failure {
   ...
   else if(s.arg1 instanceof CP && s.arg2 instanceof LSet) // cp || set,
       cp || cp
      disjCP((CP) s.arg1, (LSet) s.arg2, s);
   else if(s.arg1 instanceof LSet && s.arg2 instanceof CP){ // set || cp
      Object tmp = s.arg1;
      s.arg1 = s.arg2;
      s.arg2 = tmp;
      disjCP((CP) s.arg1, (LSet) s.arg2, s);
   }
   ...
   return;
}
```

5.5.2 Esempi d'uso

Di seguito verrà mostrato un esempio di utilizzo del vincolo di disgiunzione in un programma d'esempio.

Esempio 1

In questo esempio, si mostra un semplice funzionamento del vincolo per un `LSet` e un `CP` inizializzati:

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(1);
LSet B = LSet.empty().ins(2);
LSet C = LSet.empty().ins(new LPair(4,3)); // C = {[4,3]}
CP cp = new CP(A,B);                       // cp = {1} x {2}
solver.solve(cp.disj(C));
```

5.6 Il Vincolo di Unione

Il vincolo di unione ha la forma `union(A,B,C)`, dove `A`, `B` e `C` sono insiemi qualsiasi, e almeno uno dei 3 è di classe `CP`. Il vincolo è soddisfatto solamente se l'unione dei due insiemi `A` e `B` risulta `C`.
Il vincolo di unione è suddiviso generalmente in tre diverse casitiche generali:

1. Se A,B,C sono insiemi variabili, allora il risultato non può essere elaborato ulteriormente (e cioè, il vincolo è in forma risolta).

2. Se almeno un insieme è un prodotto non variabile, vengono definite due funzioni ausiliare $\mathcal{T}$ e $\mathcal{U}$ (vedi Capitolo 3) e l'unione viene elaborata secondo la regola di riscrittura

$$un(A, B, C) \rightarrow \mathcal{U}(A) \wedge \mathcal{U}(B) \wedge \mathcal{U}(C) \wedge un(\mathcal{T}(A), \mathcal{T}(B), \mathcal{T}(C))$$

3. Se tutti i parametri sono variabili, prodotti variabili o insiemi estensionali, si applica la regola di riscrittura per l'unione di $\mathcal{L}_{\mathcal{BR}}$ che prende un prodotto variabile come variabile.

5.6.1 Unione in JSetL

Il vincolo di unione su `CP` viene implementato nell'unico metodo `unionCP()` all'interno della classe `RwRulesSet`, suddiviso nei tre casi descritti precedentemente:

Il primo caso tratta il caso base dove tutti e tre gli argomenti sono variabili. Non essendoci quindi fondamentalmente nulla da calcolare, il vincolo è già risolto.

```
private void unionCP(LSet arg1, LSet arg2, LSet arg3, AConstraint s)
throws Failure {
   ...
   //case i
   if(arg1.isVariable() && arg2.isVariable() && arg2.isVariable()) {
      s.solved = true;
      return;
   }
```

Il secondo caso, è quello in cui almeno un insieme è un prodotto non variabile ed entrano in gioco le funzioni $\mathcal{T}$ e $\mathcal{U}$.

```
   //case ii
   if((!arg1.isVariable() && arg1 instanceof CP)
         && (!arg2.isVariable() && arg2 instanceof CP)
         && (!arg3.isVariable() && arg3 instanceof CP)) {
      s.arg1 = arg1;
      s.cons = Environment.eqCode;
      s.arg2 = arg2;
      Solver.add(new AConstraint(arg1, Environment.eqCode, arg3));
      Solver.storeInvariato = false;
      return;
   }
   if((!arg1.isVariable() && arg1 instanceof CP)
         || (!arg2.isVariable() && arg2 instanceof CP)
         || (!arg3.isVariable() && arg3 instanceof CP)) {
      LSet N1 = new LSet();
      LSet N2 = new LSet();
      LSet N3 = new LSet();
      Constraint c1 = UCP(arg1, N1);
      Constraint c2 = UCP(arg2, N2);
      Constraint c3 = UCP(arg3, N3);
      AConstraint un = new AConstraint(tauCP(arg1, N1),
                           Environment.unionCode,
```

```
                        tauCP(arg2,N2),
                        tauCP(arg3, N3));
      s.arg1 = c1.get(0).arg1;
      s.arg2 = c1.get(0).arg2;
      s.arg3 = c1.get(0).arg3;
      s.cons = c1.get(0).cons;
      Solver.add(c2);
      Solver.add(c3);
      Solver.add(un);
      Solver.storeInvariato = false;
   }
   else {
      //Volutamente vuoto
   }
   return;
   ...
}
```

Il terzo caso viene implementato indirettamente, il programma prosegue secondo le regole del linguaggio $\mathcal{L}_{\mathcal{BR}}$, ovvero trattando i `CP` come fossero `LRel/LSet` tramite ereditarietà.

La funzione ausiliaria $\mathcal{T}$, definita dal metodo `tauCP()`, è stata implementata nel seguente modo:

```
private LSet tauCP(LSet t, LSet Ni) {
   if(t instanceof CP && t.isInit() && !t.isEmpty()) {
      CP c = (CP) t;
      Object xi = c.getFirstSet().getOne();
      Object yi = c.getSecondSet().getOne();
      return Ni.ins(new LPair(xi, yi));
   }
   else if(t.isInit() && t.isEmpty())
      return LSet.empty();
   else
      return t;
}
```

La funzione ausiliaria $\mathcal{U}$, definita dal metodo `UCP()`, è stata implementata nel seguente modo:

```
private Constraint UCP(LSet t, LSet Ni) {
   if(t instanceof CP && t.isInit() && !t.isEmpty() && !t.isVariable()) {
      CP cp = (CP) t;
      Object xi = cp.getFirstSet().getOne();
      LSet Xi = cp.getFirstSet().removeOne();
      Object yi = cp.getSecondSet().getOne();
      LSet Yi = cp.getSecondSet().removeOne();

      if(!t.isEmpty() && !Xi.isEmpty() && !Yi.isEmpty()) {
         return new Constraint(new AConstraint(
            new CP(LSet.empty().ins(xi), Yi),
            Environment.unionCode,
            new CP(Xi, cp.getSecondSet().ins(yi)),
            Ni));
      }
      if(!Xi.isEmpty() && Yi.isEmpty())
         return Ni.eq(new CP(Xi, LSet.empty().ins(yi)));
      if(!Yi.isEmptySL() && Xi.isEmptySL())
         return Ni.eq(new CP(LSet.empty().ins(xi), Yi));
      if(Xi.isEmptySL() && Yi.isEmptySL())
         return Ni.eq(LSet.empty());
      return Constraint.TRUE;
   }
   return Constraint.TRUE;
}
```

5.6.2 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo del vincolo di unione in qualche programma d'esempio.

Esempio 1

In questo esempio, si mostra un semplice funzionamento del vincolo dove un `LSet` variabile viene vincolato ad essere uguale all'unione di altri due insiemi:

```
Solver solver = new Solver();
LSet A = new LSet();          // A insieme variabile
A.setName("A");
LSet B = LSet.empty();          // B insieme vuoto
LSet C = LSet.empty().ins(1);
LSet D = LSet.empty().ins(2);
CP cp = new CP("cp",C,D);           // cp = {1,2}
solver.solve(A.union(cp, B));
A.output();                         // A = {1,2}
```

5.7 Vincoli derivati

Dal momento che i vincoli di base della libreria sono ormai in grado di supportare i prodotti Cartesiani, è lecito pensare che questi vincoli possano essere combinati senza perdere di validità. Da qui i vincoli derivati, cioè quelli esprimibili come combinazione ('and' e 'or') di vincoli base, che non hanno bisogno di una particolare implementazione per supportare i `CP`, ma funzionano per compatibilità grazie ai vincoli di base.

Il vincolo di Sottoinsieme. Il primo vincolo derivato è il vincolo di sottoinsieme tra due insiemi, secondo cui uno dei due risulta contenuto all'interno del secondo seguendo la semplice regola

$$subset(s1, s2) <==> union(set1, set2, set2)$$

In JSetL, questo si traduce in:

```
private void subset(LSet set1, LSet set2, AConstraint s) throws Failure {
   if(set1.isInit() && set1.isGround() && set1 instanceof CP)
      set1 = ((CP) set1).expand();
   if(set2.isInit() && set2.isGround() && set2 instanceof CP)
      set2 = ((CP) set2).expand();
   s.arg1 = set1;
```

```
   s.arg2 = set2;
   s.arg3 = set2;
   s.arg4 = null;
   s.cons = Environment.unionCode;
   s.caseControl = 0;
   return;
}
```

Il vincoli di Intersezione. Il vincolo derivato di intersezione tra due insiemi `A` e `B`, si riferisce ad un terzo insieme `C` che risulta contenere gli elementi comuni tra `A` e `B` secondo la semplice regola

$$inters(s1, s2, s3) <==>$$
$$union(aux1.s3, s1) \wedge union(aux2, s3, s2) \wedge disj(aux1, aux2)$$

In JSetL, questo si traduce in:

```
private void inters(LSet set1, LSet set2, LSet set3, AConstraint s)
throws Failure {
   LSet aux1 = new LSet();
   s.arg1 = aux1;
   s.arg2 = set3;
   s.arg3 = set1;
   s.arg4 = null;
   s.cons = Environment.unionCode; // union(aux1,set3,set1)
   s.caseControl = 0;

   LSet aux2 = new LSet();
    // union(aux2,set3,set2)
   AConstraint c1 = new AConstraint(aux2,Environment.unionCode,set3,set2);
    // disj(aux1,aux2)
   AConstraint c2 = new AConstraint(aux1,Environment.disjCode,aux2);
   Solver.add(Solver.indexOf(s) + 1, c1);
   Solver.add(c2);
   union(aux1,set3,set1,s);
   return;
}
```

Il vincolo di Differenza. Il vincolo derivato di differenza tra due insiemi `A` e `B`, si riferisce ad un terzo insieme `C` che risulta contenere gli elementi appartenenti al primo insieme `A` e non al secondo `B` tramite la semplice regola

$$diff(s1, s2, s3) <==>$$
$$subset(s3, s1) \wedge union(s2, s3, aux1) \wedge subset(s1, aux1) \wedge disj(s2, s3)$$

In JSetL, questo si traduce in:

```
private void diff(LSet set1, LSet set2, LSet set3, AConstraint s)
throws Failure {
   LSet aux1 = new LSet();
   s.arg1 = set3;
   s.arg2 = set1;
   s.arg3 = null;
   s.arg4 = null;
   s.cons = Environment.subsetCode; // subset(s3,s1)
   s.caseControl = 0;

   // union(set2,set3,aux1)
   AConstraint c1 = new AConstraint(set2,Environment.unionCode,set3,aux1);
   // subset(s1,aux1)
   AConstraint c2 = new AConstraint(set1,Environment.subsetCode,aux1);
   // disj(s2,s3)
   AConstraint c3 = new AConstraint(set2,Environment.disjCode,set3);

   Solver.add(Solver.indexOf(s) + 1, c1);
   Solver.add(Solver.indexOf(s) + 1, c2);
   Solver.add(c3);

   subset(set3,set1,s);
   return;
}
```

Il vincolo di Rimozione. Il vincolo derivato di rimozione su un insieme `A` permette di sottrarre ad un dato insieme un elemento `e` tramite la semplice regola

$$A = A - \{e\}$$

In JSetL, questo si traduce in:

```
protected void less(AConstraint s) //
throws Failure {
   if (s.arg1 instanceof LSet && s.arg2 instanceof LPair && s.arg3
       instanceof LSet)
      less((LSet)s.arg1, (LPair)s.arg2, (LSet)s.arg3, s);
   if (s.arg1 instanceof LSet && s.arg2 instanceof LVar && s.arg3
       instanceof LSet)
      less((LSet)s.arg1, (LVar)s.arg2, (LSet)s.arg3,s);
   else if (s.arg3 instanceof LSet) {
      if (s.arg1 instanceof Set) {
         LSet aux = new LSet((Set<?>)s.arg1);
         s.arg1 = aux;
      }
      if (!(s.arg2 instanceof LVar)) {
         LVar aux = new LVar(s.arg2);
         s.arg2 = aux;
      }
      less(s);
   }
   return;
}
```

5.7.1 Esempi d'uso

Di seguito verranno mostrati alcuni esempi di utilizzo dei vincoli derivati in qualche semplice programma d'esempio.

Esempio 1

In questo esempio, mostriamo un semplice funzionamento del vincolo di Sottoinsieme, dove un `LSet` completamente specificato viene vincolato ad essere un sottoinsieme di un `CP`:

```
   Solver solver = new Solver();
   LSet A = LSet.empty().ins(2);
   LSet B = LSet.empty().ins(1).ins(2);
```

```
LSet C = LSet.empty().ins(new LPair(2,1));
CP cp = new CP(A,B);
solver.solve(C.subset(cp));
```

Esempio 2

In questo esempio, mostriamo un semplice funzionamento del vincolo di Intersezione, dove un `LSet` viene vincolato ad essere l'intersezione tra un `CP` e un `LSet`, esplorando le soluzioni possibili:

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(2);
LSet B = LSet.empty().ins(1).ins(2);
LSet D = LSet.empty().ins(new LPair(2,1));
CP cp = new CP(A,B);
LSet C = new LSet();
solver.solve(C.inters(cp, D)); // INSIEME VUOTO
solver.nextSolution();     // COPPIA [1,2]
solver.nextSolution();     // COPPIA [1,2]
solver.nextSolution();     // UNKNOWN
```

Esempio 3

In questo esempio, mostriamo un semplice funzionamento del vincolo di Differenza, dove viene sostanzialmente posta l'equazione $cp = C - D$ dove cp è `CP` noto, C è un `LSet` noto e D è un `variabile` con l'obiettivo di dare un valore a D:

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(1);
LSet B = LSet.empty().ins(3).ins(2);
LSet C = LSet.empty().ins(new LPair(1,2));
CP cp = new CP(A,B);
LSet D = new LSet();
solver.solve(cp.diff(C,D));   // D = [1,3]
```

Esempio 4

In questo esempio, mostriamo un semplice funzionamento del vincolo di Rimozione di una coppia da un `CP`:

```
Solver solver = new Solver();
LSet A = LSet.empty().ins(1);
LSet B = LSet.empty().ins(3).ins(2);
CP cp = new CP(A,B);          // cp = {1} x {2,3}
LSet D = new LSet();
solver.solve(cp.less(new LPair(1,2), D)); // D = {[1,3]}
```

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come integrare la nozione di prodotto Cartesiano, con i relativi vincoli su `CP`, all'interno della libreria Java JSetL. La presenza in JSetL degli insiemi logici (classe LSet) ed i vincoli ad essi associati ha permesso l'integrazione dei prodotti Cartesiani all'interno della libreria sfruttando la struttura interna di rappresentazione e manipolazione degli insiemi logici, aggiungendo opportune classi e vincoli in grado di supportare la nuova struttura dati implementata dalla classe `CP`.

Come sappiamo, il prodotto Cartesiano non è altro che l'insieme delle coppie ordinate di due insiemi. Una delle applicazioni informatiche più comuni di questo concetto si trova nel campo dei database, dove il prodotto Cartesiano dei dati è il modo naturale per unire due set differenti. La sua caratteristica risiede nel modo di determinare la griglia: è sufficiente conoscerne gli elementi degli assi.
In realtà le applicazioni del prodotto cartesiano sono molteplici. L'esempio più chiaro è nella creazione di grafici e report, ovvero ogni qual volta la combinazione di due valori deve dare un'unica coppia.

Per quando riguarda futuri sviluppi, un aspetto sicuramente da migliorare riguarda l'implementazione delle regole di riscrittura aggiuntive per coprire

tutti quei casi ancora non trattati, come già anticipato all'inizio del capitolo 2.2. Un altro aspetto importante è permettere (eventualmente) un miglioramento in efficienza, poichè normalmente "calcolare" un prodotto Cartesiano risulta computazionalmente costoso, e bisogna cercare, mediante opportune euristiche, di calcolarlo il meno possibile e solo quando serve veramente. Infine un ulteriore sviluppo è quello di migliorare l'integrazione tra i prodotti cartesiani e altre astrazioni aggiunte di recente a JSetL (come i cosiddetti Restricted Intensional Sets).

Bibliografia

[1] Maximiliano Cristià, Gianfranco Rossi. *A set solver for finite relation algebra*. In Relational and Algebraic Methods in Computer Science (RAMICS 2018). LECTURE NOTES IN COMPUTER SCIENCE, vol. 11194, p. 333-349, Springer, 2018

[2] Maximiliano Cristiá and Gianfranco Rossi. *A Decision Procedure for Sets, Binary Relations and Partial Functions*, in Computer Aided Verification - 28th International Conference (CAV 2016), Lecture Notes in Computer Science, Vol. 9779, pages 179-198, Springer, 2016.

[3] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. *Sets and Constraint Logic Programming.* ACM Transaction on Programming Language and Systems, Vol. 22(5), Sept.2000, 861-931

[4] G. Rossi. {`log`} home-page. http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html

[5] G. Rossi, E. Panegai, and E. Poleo. *JSetL; a Java Library for Supporting Declarative Programming in Java.* Software-Practice & Experience (ISSN: 0038-064), 37:115-149,2007.

[6] G. Rossi JSetL home-page. http://cmt.math.unipr.it/jsetl.html

[7] Michael Cobianchi. *Trattamento di vincoli su insiemi e relazioni binarie nella libreria Java JSetL.* Tesi di Laurea del Corso di Studio in Informatica, Università di Parma, Dicembre 2018.

[8] Andrea Fois. *Estensioni ed uso degli Insiemi Intensionali Ristretti in JSetL.* Tesi di Laurea del Corso di Studio in Informatica, Università di Parma, Luglio 2018.