



UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE,
FISICHE E INFORMATICHE

CORSO DI LAUREA IN INFORMATICA

Fattorizzazione cartesiana nella libreria PPLite

Cartesian Factoring in the PPLite Library

Laureando:

Luigi ZACCONE

Relatore:

Prof. Enea ZAFFANELLA

Anno Accademico 2017/2018

INDICE

1	INTRODUZIONE	4
2	FATTORIZZAZIONE CARTESIANA	6
2.1	Blocchi e fattori	7
2.2	Partizione ammissibile	9
2.3	Operazioni	9
2.3.1	Inclusion Test (\sqsubseteq)	12
2.3.2	Join (\sqcup)	13
2.3.3	Meet (\sqcap)	14
2.3.4	Conditional	15
2.3.5	Assignment	16
2.3.6	Widening (∇)	17
3	IMPLEMENTAZIONE	19
3.1	F_Poly	20
3.1.1	check_inv	21
3.2	Operazioni di base	24
3.2.1	refactor	24
3.2.2	least_upper_bound	26
3.2.3	merge	27
3.3	Operazioni sui poliedri	28
3.3.1	inclusion	28
3.3.2	join	29
3.3.3	add_con	31
3.3.4	affine_image e affine_preimage	33
3.4	Altre operazioni	34
3.4.1	add_space_dims e remove_space_dims	34
3.4.2	add_gen	37

3.4.3	Predicati	38
3.4.4	Overload degli operatori	39
4	VALUTAZIONE SPERIMENTALE	41
5	CONCLUSIONE	43
	BIBLIOGRAFIA	44

1 INTRODUZIONE

I poliedri sono sottoinsiemi di uno spazio n -dimensionale, definiti come intersezione di un insieme finito di semispazi. Hanno un'ottima capacità di rappresentare legami e proprietà tra n elementi con valori nel reale, trovando applicazione nell'analisi di sistemi complessi. In particolare, nell'*interpretazione astratta* sono utilizzati in campo informatico per l'analisi statica di codice, ovvero riuscire a individuare a *compile time* errori di overflow o possibili accessi errati in memoria utilizzando strutture dati (come array o vettori). L'utilizzo di questi strumenti è però limitato da problemi di complessità computazionale, con caso pessimo di tipo esponenziale. I tentativi per diminuirne la complessità si sono concentrati nel trovare domini meno complessi e con un inferiore peso computazionale, ma che mantenessero comunque una buona precisione (e.g. pentagons, octagons, bounded differences). È stata provata anche l'implementazione di nuovi algoritmi che permettessero una maggiore efficienza avendo però dei risultati approssimati.

Le implementazioni di questo dominio sono comunque in continua evoluzione. Fino ad'ora se ne possono distinguere diverse in base al modo di rappresentare i poliedri. Un esempio è la rappresentazione *singola* (che utilizza solamente vincoli) o la *doppia rappresentazione* (descritta nella sezione successiva, che utilizza sia vincoli che generatori).

L'idea di base, proposta in [5] e in seguito migliorata leggermente in [6] è quella di dividere i poliedri in una serie di sotto poliedri di dimensione inferiore in modo da eseguire più efficientemente le operazioni su di essi. I sotto poliedri saranno formati dai vincoli che hanno variabili in comune.

La *PPL*, in dettaglio descritta in [2], è una libreria scritta in C++ utilizzata per la manipolazione numerica di informazioni. La *PPLite*, esposta in [4] e in [3], si ispira al lavoro della *PPL* ma con obiettivi differenti: mettere a disposizione a ricercatori e studenti un framework *leggero*, scritto in C++ *moderno*,

per la sperimentazioni di idee e algoritmi nel contesto della computazione dei poliedri.

Il lavoro svolto è stato quello di, inizialmente, effettuare un porting della *fattorizzazione cartesiana* dalla PPL, descritta in [1], alla PPLite. Una volta effettuato il porting si è provveduto a modernizzare il codice e a implementare nuove operazioni.

Il resto della tesi è organizzato nel modo seguente:

- nel capitolo 1 verranno introdotte, formalmente, le funzioni principali descritte in [6];
- nel capitolo 2 verranno descritte le implementazioni concrete di suddette funzioni e anche di alcune funzioni helper e aggiuntive;
- nel capitolo 3 verranno mostrati alcuni test di correttezza e un test di efficienza;
- nella conclusione verrà discusso il lavoro svolto e proposti sviluppi futuri.

2 FATTORIZZAZIONE CARTESIANA

I poliedri chiusi convessi che saranno utilizzati potranno essere rappresentati in due modi:

- tramite un sistema finito di vincoli \mathcal{C} . I vincoli non sono altro che le equazioni e disequazioni che individuano rispettivamente gli iperpiani e i semispazi del poliedro [3];
- tramite un sistema finito di generatori \mathcal{G} . Questi permettono di costruire il luogo geometrico dei punti determinati da combinazioni di particolari elementi del poliedro stesso [3]. I generatori possono essere formati da:
 - **punto**: ogni elemento del poliedro \mathcal{P} è un suo punto. Un punto può anche essere un *vertice* se non può essere espresso come combinazione convessa di altri punti in \mathcal{P} ;
 - **raggio**: un raggio di un poliedro non vuoto \mathcal{P} è un vettore $\mathbf{r} \in \mathbb{R}^n$ non nullo tale che per ogni $\mathbf{x} \in \mathcal{P}$ e per ogni $\mu \geq 0$ vale

$$(\mathbf{x} + \mu \mathbf{r}) \in \mathcal{P}$$

se $\mathcal{P} = \emptyset$ allora non ha raggi.

Un raggio quindi definisce una direzione dove il poliedro è illimitato.

- **retta**: un vettore $\mathbf{l} \in \mathbb{R}^n$ è una retta di un poliedro non vuoto \mathcal{P} se e solo se i vettori \mathbf{l} e $-\mathbf{l}$ sono entrambi raggi del poliedro \mathcal{P} . Vale quindi che per ogni $\mathbf{x} \in \mathcal{P}, \mu \in \mathbb{R}$

$$(\mathbf{x} + \mu \mathbf{l}) \in \mathcal{P}$$

Se $\mathcal{P} = \emptyset$ allora non ha rette.

Quindi possiamo esprimere un generatore come $\mathcal{G} = (L, R, P)$ dove L, R, P sono sottoinsiemi finiti di \mathbb{R}^n di cardinalità l, r e p rispettivamente, tali che $\mathbf{0} \notin R$ e $\mathbf{0} \notin L$, con

$$\mathcal{P} = \text{gen}(\mathcal{G}) \stackrel{\text{def}}{=} \{L\boldsymbol{\lambda} + R\boldsymbol{\rho} + P\boldsymbol{\pi} \mid \boldsymbol{\lambda} \in \mathbb{R}^l, \boldsymbol{\rho} \in \mathbb{R}_+^r, \boldsymbol{\pi} \in \mathbb{R}_+^p, \sum_{i=1}^p \pi_i = 1\}$$

I punti di un poliedro \mathcal{P} sono quindi ottenibili come somma di combinazioni convesse dei punti in P e di combinazioni non negative dei raggi in R (e quindi combinazioni lineari delle rette in L).

La fattorizzazione cartesiana, quando applicata, permette di diminuire la complessità di esecuzione delle operazioni sui poliedri convessi. L'idea è che i poliedri utilizzati nell'analisi dei programmi non mettono in relazione tutte le variabili del programma in un solo vincolo. Per esempio: un ipercubo $H_n = \{0 \leq x_i \leq 1 \mid i = 1, \dots, n\}$ richiede 2^n generatori di n dimensioni per essere rappresentato, tramite la decomposizione invece ne sono necessari solo $2n$ di dimensione 1. La terminologia e gli esempi saranno ispirati al lavoro svolto in [6].

2.1 BLOCCHI E FATTORI

Qui e nei successivi paragrafi verranno considerati esclusivamente poliedri *non* vuoti.

Sia $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ un insieme di n variabili. Dato un poliedro, \mathcal{X} può essere partizionato in un sottoinsieme \mathcal{X}_k che chiamiamo *blocco* tale che i vincoli esistono solo tra variabili presenti nello stesso *blocco*. Quindi, ogni variabile priva di vincoli risiede in un singoletto. Ci riferiamo a questo insieme come $\pi = \pi_P = \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_t\}$.

Esempio 2.1.1. Consideriamo

$$\begin{aligned} \mathcal{X} &= \{x_1, x_2, x_3\} \text{ e} \\ P &= \{x_1 + 2x_2 \leq 3\}. \end{aligned}$$

In questo caso \mathcal{X} viene partizionato in due blocchi: $\mathcal{X}_1 = \{x_1, x_2\}$ e $\mathcal{X}_2 = \{x_3\}$.

Per ogni blocco, quindi, sarà presente un *fattore* P_k che sarà composto solo dalle variabili presenti nel suo blocco. In qualsiasi momento il poliedro originale può essere recuperato "concatenando" i fattori, che equivale ad applicare l'unione dei sistemi di vincoli \mathcal{C}_{P_k} e una variante del prodotto cartesiano (in quanto viene applicato solo ai punti e non alle linee o ai raggi) dei generatori \mathcal{G}_{P_k} .

Esempio 2.1.2. Consideriamo un poliedro \mathcal{P} con i seguenti vincoli e generatori:

$$\mathcal{C} = \{-x_1 \leq -1, x_1 \leq 4, -x_2 \leq -2, x_2 \leq 4\}$$

$$\mathcal{G} = \{L, R, (1, 2)^T, (1, 4)^T, (4, 2)^T, (4, 4)^T\}$$

Con L insieme delle rette e R insieme dei raggi.

Il poliedro non ha vincoli tra le variabili x_1 e x_2 . Quindi, $\mathcal{X} = \{x_1, x_2\}$ può essere partizionato nei blocchi: $\pi_P = \{\{x_1\}, \{x_2\}\}$ con i risultanti fattori $P_1 = (\mathcal{C}_{P_1}, \mathcal{G}_{P_1})$ e $P_2 = (\mathcal{C}_{P_2}, \mathcal{G}_{P_2})$ dove:

$$\mathcal{C}_{P_1} = \{-x_1 \leq -1, x_1 \leq 4\}$$

$$\mathcal{C}_{P_2} = \{-x_2 \leq -2, x_2 \leq 4\}$$

$$\mathcal{G}_{P_1} = \{L, R, (1, 4)^T\}$$

$$\mathcal{G}_{P_2} = \{L, R, (2, 4)^T\}$$

Il poliedro originale può essere ricavato da P_1 e P_2 come $P = P_1 \bowtie P_2 = (\mathcal{C}_{P_1} \cup \mathcal{C}_{P_2}, \mathcal{G}_{P_1} \times \mathcal{G}_{P_2})$

L'insieme \mathcal{L} che consiste in tutte le partizioni possibili di \mathcal{X} forma un *reticolo di partizioni* $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$. Gli elementi π del reticolo sono ordinati come segue: $\pi \sqsubseteq \pi'$, se ogni blocco di π è incluso in qualche blocco di π' , si dice quindi che π è "più fine" di π' o, equivalentemente che π' è più "grossolana" di π . Questo reticolo è dotato degli operatori di *least upper bound* (\sqcup) che permette di calcolare, dati due blocchi, la partizione più raffinata e *greatest upper bound* (\sqcap) che permette di calcolarne quella più grossolana. È da notare che nel reticolo della partizione $\top = \{\mathcal{X}\}$ e $\perp = \{\{x_1\}, \{x_2\}, \dots, \{x_n\}\}$.

2.2 PARTIZIONE AMMISSIBILE

Una partizione π è *ammissibile* per un poliedro P se non esistono variabili x_i e x_j in diversi blocchi di π relazionati da un vincolo di P , ovvero $\pi \sqsupseteq \pi_P$. Le partizioni ammissibili sono un sottoinsieme chiuso superiormente del reticolo, chiameremo questo sottoinsieme \mathcal{B} . Se $\pi \in \mathcal{B}$ allora lo saranno anche tutte le π' tali che $\pi, \pi' \in \mathcal{L}$ e $\pi \sqsubseteq \pi'$. Le partizioni ammissibili sono dotate di elemento minimo ovvero la più fine partizione appartenente a \mathcal{B} , partizione che verrà anche considerata come la migliore. Questa partizione minima viene calcolata come spiegato nell'**esempio 2.1.1**.

Il nostro obiettivo è quello di cercare di utilizzare sempre la partizione ammissibile minima.

2.3 OPERAZIONI

In questo paragrafo verrà spiegato l'effetto che alcune operazioni dei poliedri generano sui fattori. Occorre premettere che in quasi tutte le operazioni binarie, i poliedri hanno la stessa dimensione. Fa eccezione la concatenazione. Per descrivere meglio queste operazioni, è necessario prima dare una specifica di alcune funzioni utilizzate all'interno delle prime.

Convert Constraint. Funzione che mappa le dimensioni di un vincolo o generatore all'interno di un fattore. La PPLite utilizza dimensioni nel range $[0, \dots, n]$ per i poliedri. Tramite la fattorizzazione è quindi necessario tradurre gli indici esterni a quelli interni dei vari blocchi.

Least Upper Bound. Questa funzione estrae il *lub* di due partizioni. Dall'Algoritmo 1 si può notare come sia necessario generare l'insieme delle unioni dei due blocchi π_P e π_Q in π . Successivamente si uniscono tra loro tutti i blocchi di π che hanno un'intersezione non vuota, calcolando successivamente la partizione corretta.

Algoritmo 1 Least-Upper-Bound

```

1: function LEAST-UPPER-BOUND( $\pi_{\mathcal{P}}, \pi_{\mathcal{Q}}$ )
2:    $\pi := \emptyset$ 
3:   for each  $\mathcal{X}_{\mathcal{P}_i}$  in  $\pi_{\mathcal{P}}$  do
4:      $\mathcal{B} := \emptyset$ 
5:     for each  $\mathcal{X}_{\mathcal{Q}_k}$  in  $\pi_{\mathcal{Q}}$  do
6:       if  $\mathcal{X}_{\mathcal{P}_i} \cap \mathcal{X}_{\mathcal{Q}_k} \neq \emptyset$  then
7:          $\mathcal{B} := \mathcal{B} \cup \mathcal{X}_{\mathcal{Q}_k}$ 
8:      $\pi.\text{add}(\mathcal{B})$ 
9:   while  $\exists \mathcal{X}_i, \mathcal{X}_j \in \pi, i \neq j$  t.c.  $\mathcal{X}_i \cap \mathcal{X}_j \neq \emptyset$  do
10:     $\pi := (\pi \setminus \{\mathcal{X}_i, \mathcal{X}_j\}) \cup \{\mathcal{X}_i \cup \mathcal{X}_j\}$ 
    return  $\pi$ 

```

Merge. Definito come \uparrow , quando si ha l'operazione $\pi \uparrow \mathcal{A}$, dove π è una partizione di un poliedro e \mathcal{A} è un sottoinsieme delle sue variabili (non per forza connesse da vincoli), genera una partizione π_m tale che:

- $\exists \mathcal{X}_i \in \pi_m : \mathcal{A} \subseteq \mathcal{X}_i, \pi \sqsubseteq \pi_m$

Viene quindi generata una nuova fattorizzazione unendo i blocchi del poliedro che hanno variabili in comune con \mathcal{A} . Questa unione genera un blocco unico \mathcal{D} , tale che $\mathcal{A} \subseteq \mathcal{B}$.

Algoritmo 2 Merge

```

1: function MERGE( $\pi, \mathcal{A}$ )
2:    $\pi_m := \emptyset$ 
3:    $\mathcal{B} := \emptyset$ 
4:   for each  $\mathcal{X}_i$  in  $\pi$  do
5:     if  $\mathcal{X}_i \cap \mathcal{A} \neq \emptyset$  then
6:        $\mathcal{B} := \mathcal{B} \cup \mathcal{X}_i$ 
7:     else
8:        $\pi_m.\text{add}(\mathcal{X}_i)$ 
9:    $\pi_m.\text{add}(\mathcal{B})$ 
10:  return  $\pi_m$ 

```

Refactor. Questa operazione ha come argomenti un poliedro fattorizzato e una fattorizzazione ammissibile π per questo poliedro. Non fa altro che convertire i fattori rispetto alla seconda fattorizzazione. È importante notare che la partizione ammissibile non sarà più fine di $\pi_{\mathcal{P}}$, visto che i blocchi non vengono divisi bensì solo uniti.

La refactor è un'operazione necessaria in quanto, avendo due poliedri \mathcal{P} e \mathcal{Q} definiti sullo stesso insieme di variabili $X = \{x_1, x_2, \dots, x_n\}$ e presi $\pi_{\mathcal{P}} = \{X_{\mathcal{P}_1}, X_{\mathcal{P}_2}, \dots, X_{\mathcal{P}_r}\}$, $\pi_{\mathcal{Q}} = \{X_{\mathcal{Q}_1}, X_{\mathcal{Q}_2}, \dots, X_{\mathcal{Q}_s}\}$ partizioni corrette di \mathcal{P} e \mathcal{Q} in genere abbiamo che $\pi_{\mathcal{P}} \neq \pi_{\mathcal{Q}}$. È necessario che, come in molte operazioni comuni nell'analisi e manipolazione dei poliedri, le dimensioni delle partizioni utilizzate siano uguali in quantità e valori. Se si applica la fattorizzazione su poliedri aventi gli stessi blocchi non si fa altro che calcolare il *lub* ($\pi = \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$).

Algoritmo 3 Refactor

```

1: function REFACTOR( $\mathcal{P}, \pi, \pi_{\mathcal{P}}$ )
2:    $\mathcal{O}, \pi_{\mathcal{O}} := \emptyset$ 
3:   for each  $i$  in  $\{1, \dots, r\}$  do                                 $\triangleright r = \text{numero blocchi di } \pi$ 
4:      $\mathcal{O}_i, \pi_{\mathcal{O}_i} := \emptyset$ 
5:     for each  $j$  in  $\{1, \dots, m\}$  do                                 $\triangleright m = \text{numero blocchi di } \pi_{\mathcal{P}}$ 
6:       if  $\pi_{\mathcal{P}_i} \cap \pi_j \neq \emptyset$  then
7:          $\mathcal{O}_i := \mathcal{O}_j \bowtie \mathcal{P}_i$ 
8:          $\pi_{\mathcal{O}_i} := \pi_{\mathcal{O}_i} \bowtie \pi_{\mathcal{P}_i}$ 
9:        $\mathcal{O}.\text{add}(\mathcal{O}_i)$ 
10:       $\pi_{\mathcal{O}}.\text{add}(\pi_{\mathcal{O}_i})$ 
11:   REMAP-DIMENSIONS( $\mathcal{O}, \pi_{\mathcal{O}}, \pi$ )
12:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 

```

Come spiegato precedentemente, la refactor prende in input un poliedro fattorizzato \mathcal{P} e la sua partizione π per rifattorizzarlo in base al terzo parametro $\pi_{\mathcal{P}}$. Non fa altro che unire i fattori che hanno intersezione dei relativi blocchi non vuota tra $\pi_{\mathcal{P}}$ e π e generare il blocco finale. È importante ri-mappare le dimensioni dei fattori di output in base a quelle descritte nel blocco di $\pi_{\mathcal{P}}$.

2.3.1 INCLUSION TEST (\sqsubseteq)

Operazione necessaria per controllare se un poliedro è incluso in un altro. Disponendo di una doppia rappresentazione è possibile controllarlo se, dati due poliedri \mathcal{P} e \mathcal{Q} , tutti i generatori in $\mathcal{G}_{\mathcal{P}}$ soddisfano tutti i vincoli in $\mathcal{C}_{\mathcal{Q}}$.

Nel nostro caso, come mostrato nell'Algoritmo 4, è stata implementata tramite wrapper. In particolare vengono rifattorizzati i poliedri con il *lub* $\pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ e successivamente viene applicata l'*inclusion* in ordine su ogni coppia di fattori: il test avrà successo se tutti i test ritornano *True*.

Algoritmo 4 Inclusion Test

```

1: function INCLUSION( $\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{Q}, \pi_{\mathcal{Q}}$ )
2:    $\pi := \text{LEAST-UPPER-BOUND}(\pi_{\mathcal{P}}, \pi_{\mathcal{Q}})$ 
3:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
4:    $\mathcal{Q}' := \text{REFACTOR}(\mathcal{Q}, \pi_{\mathcal{Q}}, \pi)$ 
5:   for each  $k$  in  $\{1, \dots, r\}$  do                                 $\triangleright r = \text{numero di blocchi in } \pi$ 
6:     if  $\mathcal{P}'_k$  non include  $\mathcal{Q}'_k$  then
7:       return False
8:   return True

```

2.3.2 JOIN (\sqcup)

Tramite la doppia rappresentazione, i generatori $\mathcal{G}_{\mathcal{O}}$ dove \mathcal{O} è il risultato del *join*, sono semplicemente l'unione dei generatori dei poliedri presi in input dalla funzione, ovvero $\mathcal{G}_{\mathcal{O}} = \mathcal{G}_{\mathcal{P}} \cup \mathcal{G}_{\mathcal{Q}}$. I vincoli $\mathcal{C}_{\mathcal{O}}$ sono ottenuti aggiungendo incrementalmente i generatori di $\mathcal{G}_{\mathcal{Q}}$ al poliedro definito da $\mathcal{C}_{\mathcal{P}}$.

Algoritmo 5 Join

```

1: function JOIN( $\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{O}, \pi_{\mathcal{O}}$ )
2:   if IS_EMPTY( $\mathcal{P}$ ) then
3:      $\mathcal{O}, \pi_{\mathcal{O}} := \mathcal{Q}, \pi_{\mathcal{Q}}$ 
4:   else if IS_EMPTY( $\mathcal{Q}$ ) then
5:      $\mathcal{O}, \pi_{\mathcal{O}} := \mathcal{P}, \pi_{\mathcal{P}}$ 
6:   else
7:      $\mathcal{O}, \pi_{\mathcal{O}} := \text{JOIN-POLY}(\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{O}, \pi_{\mathcal{O}})$ 
8:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 

```

Avendo un sistema di poliedri fattorizzati, è necessario per prima cosa ri-fattorizzare \mathcal{P} e \mathcal{Q} utilizzando il loro *lub*. Per ogni coppia di fattori $(\mathcal{P}'_i, \mathcal{Q}'_i)$ uguali, se ne aggiunge uno con il rispettivo blocco π_i al risultato. Se invece i due fattori risultano diversi, ognuno viene unita a una coppia di fattori comuni $(\mathcal{P}_{\mathcal{T}}, \mathcal{Q}_{\mathcal{T}})$ e successivamente si calcola la *join* di questi due fattori e si inserisce insieme al rispettivo blocco nel poliedro di output.

Un caso speciale è stato assegnato ai poliedri *vuoti*. Dati \mathcal{P} come poliedro *vuoto* e \mathcal{Q} un poliedro generico allora $\mathcal{Q} \sqcup \mathcal{P} = \mathcal{Q}$ e anche $\mathcal{P} \sqcup \mathcal{Q} = \mathcal{Q}$.

Se invece $\pi = \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ e $\mathcal{U} = \{\mathcal{X}_k \mid \mathcal{P} = \mathcal{Q}, \mathcal{X}_k \in \pi\}$ allora la partizione ammissibile, in questo caso, sarà uguale a:

$$\pi_{\mathcal{P} \sqcup \mathcal{Q}} = \mathcal{U} \cup \bigcup_{\mathcal{T} \in \pi \setminus \mathcal{U}} \mathcal{T}$$

Algoritmo 6 Join-Poly

```

1: function JOIN-POLY( $\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{O}, \pi_{\mathcal{O}}$ )
2:    $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ 
3:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
4:    $\mathcal{Q}' := \text{REFACTOR}(\mathcal{Q}, \pi_{\mathcal{Q}}, \pi)$ 
5:   for each  $i$  in  $\{0, \dots, |\mathcal{P}'|\}$  do
6:     if  $\mathcal{P}'_i = \mathcal{Q}'_i$  then
7:        $\mathcal{O}.\text{add}(\mathcal{P}'_i)$ 
8:        $\pi_{\mathcal{O}}.\text{add}(\mathcal{X}'_i)$ 
9:     else
10:       $\mathcal{X}_{\mathcal{T}} := \mathcal{X}_{\mathcal{T}} \cup \mathcal{X}_i$ 
11:       $\mathcal{P}_{\mathcal{T}} := \mathcal{P}_{\mathcal{T}} \bowtie \mathcal{P}'_i$ 
12:       $\mathcal{Q}_{\mathcal{T}} := \mathcal{Q}_{\mathcal{T}} \bowtie \mathcal{Q}'_i$ 
13:    $\mathcal{P}_{\mathcal{T}} := \mathcal{P}_{\mathcal{T}} \sqcup \mathcal{Q}_{\mathcal{T}}$ 
14:    $\pi_{\mathcal{O}}.\text{add}(\mathcal{X}_{\mathcal{T}})$ 
15:    $\mathcal{O}.\text{add}(\mathcal{P}_{\mathcal{T}})$ 
16:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 
    
```

2.3.3 MEET (\sqcap)

Per la doppia rappresentazione, $\mathcal{P} \sqcap \mathcal{Q}$ genera un poliedro i cui vincoli $\mathcal{C}_{\mathcal{P} \sqcap \mathcal{Q}}$ sono risultati dall'unione di $\mathcal{C}_{\mathcal{P}}$ e $\mathcal{C}_{\mathcal{Q}}$, mentre $\mathcal{G}_{\mathcal{P} \sqcap \mathcal{Q}}$ si ottiene aggiungendo incrementalmente i vincoli di $\mathcal{C}_{\mathcal{Q}}$ al poliedro \mathcal{P} . Se $\mathcal{P} \sqcap \mathcal{Q}$ risulta non soddisfacibile allora $\mathcal{P} \sqcap \mathcal{Q} = \perp$ e $\mathcal{G}_{\mathcal{P} \sqcap \mathcal{Q}} = \emptyset$.

Per quanto riguarda i poliedri fattorizzati è necessario generare $\pi = \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ e rifattorizzare i due poliedri \mathcal{P} e \mathcal{Q} tramite essa, generando quindi \mathcal{P}' e \mathcal{Q}' . Per tutte le r coppie di fattori \mathcal{P}'_i e \mathcal{Q}'_i , se sono uguali aggiungo \mathcal{P}'_i con il rispettivo blocco al poliedro di output \mathcal{O} , altrimenti creo un fattore $\mathcal{F} = \mathcal{P}'_i \sqcap \mathcal{Q}'_i$. Se \mathcal{F} dovesse risultare vuoto, allora il risultato dell'intera operazione di *meet* è un poliedro \perp . Se al contrario non fosse vuoto, aggiungo \mathcal{F} a \mathcal{O} per poi riprendere il ciclo fino all'esaurimento dei fattori e ritornando infine \mathcal{O} e il suo blocco (rappresentato dal *lub* di $\pi_{\mathcal{P}}$ e $\pi_{\mathcal{Q}}$) come risultato. $\pi_{\mathcal{P} \sqcup \mathcal{Q}} = \pi_{\mathcal{P}} \sqcap \pi_{\mathcal{Q}}$ è una partizione ammissibile se $\mathcal{P} \sqcap \mathcal{Q} \neq \perp$, altrimenti \perp è ammissibile.

Algoritmo 7 Meet

```

1: function MEET( $\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{Q}, \pi_{\mathcal{Q}}$ )
2:    $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ 
3:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
4:    $\mathcal{Q}' := \text{REFACTOR}(\mathcal{Q}, \pi_{\mathcal{Q}}, \pi)$ 
5:    $\mathcal{O} := \emptyset$ 
6:   for each  $i$  in  $\{1, \dots, r\}$  do                                 $\triangleright r = \text{numero di blocchi in } \pi$ 
7:     if  $\mathcal{P}'_i = \mathcal{Q}'_i$  then
8:        $\mathcal{O}.\text{add}(\mathcal{P}'_i)$ 
9:     else
10:       $\mathcal{F} := \mathcal{P}'_i \sqcap \mathcal{Q}'_i$ 
11:      if IS_EMPTY( $\mathcal{F}$ ) then
12:        return  $\perp, \perp$ 
13:       $\mathcal{O}.\text{add}(\mathcal{F})$ 
14:    $\pi_{\mathcal{O}} := \pi$ 
15:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 
    
```

2.3.4 CONDITIONAL

Questa operazione viene utilizzata per aggiungere nuovi vincoli tra le variabili di un poliedro. Tramite la doppia rappresentazione è possibile aggiungendo un vincolo arbitrario c all'insieme $\mathcal{C}_{\mathcal{P}}$. Se dopo l'inserimento il sistema di vincoli risulta insoddisfacibile allora il poliedro diventa vuoto. Il sistema di generatori

è, come sempre, ricavato dall'aggiunta incrementale del vincolo c nel poliedro attraverso la conversione.

Utilizzando poliedri fattorizzati è necessario ricavare il blocco \mathcal{B} che contiene le variabili utilizzate nel vincolo, rifattorizzando \mathcal{P} con $\pi_{\mathcal{P}} \uparrow \mathcal{B}$ si genera \mathcal{P}' . Successivamente bisogna prendere il fattore relativo contenente le variabili di \mathcal{B} , convertire il vincolo con le dimensioni *interne* del fattore e aggiungerlo a quest'ultimo utilizzando un'operazione per normali poliedri. Si controlla, infine, che il sistema di vincoli sia ancora soddisfacibile e, in caso negativo i blocchi e i fattori vengono modificati in \perp rendendo il poliedro *vuoto*. È importante far notare che, dato \mathcal{O} un poliedro risultante dall'aggiunta di un vincolo, $\pi_{\mathcal{O}} = \pi_{\mathcal{P}} \uparrow \mathcal{B}$ è ammissibile se $\mathcal{O} \neq \perp$, altrimenti $\pi_{\mathcal{O}} = \perp$.

Algoritmo 8 Conditional

```

1: function CONDITIONAL( $con$ )
2:    $\mathcal{B} := \text{EXTRACT\_BLOCK}(\mathcal{P}, \pi_{\mathcal{P}}, con)$ 
3:    $\pi := \pi_{\mathcal{P}} \uparrow \mathcal{B}$ 
4:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
5:    $\mathcal{O} := \emptyset$ 
6:   for each  $i$  in  $\{1, \dots, r\}$  do                                 $\triangleright r = \text{numero di blocchi in } \pi$ 
7:     if  $\mathcal{B} \cap \pi_i \neq \emptyset$  then
8:        $con_{int} := \text{CONVERT\_CON}(\pi_i, con)$ 
9:        $\mathcal{F} := \text{ADD\_CON}(\mathcal{P}'_i, con_{int})$ 
10:       $\mathcal{O}.\text{add}(\mathcal{F})$ 
11:      if IS_EMPTY( $\mathcal{O}_i$ ) then
12:        return  $\perp, \perp$ 
13:      else
14:         $\mathcal{O}.\text{add}(\mathcal{P}'_i)$ 
15:    $\pi_{\mathcal{O}} := \pi$ 
16:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 

```

2.3.5 ASSIGNMENT

Per quanto riguarda il dominio dei poliedri fattorizzati, l'operazione di *assignment* è molto simile alla *conditional*. Dopo aver estratto il blocco \mathcal{B} che indica

le variabili utilizzate nell'espressione lineare, si fattorizza il poliedro \mathcal{P} con $\pi_{\mathcal{P}} = \pi_{\mathcal{P}} \uparrow \mathcal{B}$ ottenendo \mathcal{P}' , successivamente si prende il blocco con le variabili di \mathcal{B} e si applica al relativo fattore un *assignment* come per i poliedri non fattorizzati, rimappando correttamente le variabili esterne rispetto a quelle interne al fattore. La partizione ammissibile per un poliedro risultante da questa operazione è $\pi_{\mathcal{P}} \uparrow \mathcal{B}$.

Algoritmo 9 Assignment

```

1: function ASSIGNMENT( $\mathcal{P}, \pi_{\mathcal{P}}, stmt$ )
2:   let  $stmt = (x_i = ax + \epsilon)$ 
3:    $\mathcal{B} := \text{EXTRACT\_BLOCK}(stmt)$ 
4:    $\pi := \pi_{\mathcal{P}} \uparrow \mathcal{B}$ 
5:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
6:    $\mathcal{O} := \emptyset$ 
7:   for each  $i$  in  $\{1, \dots, r\}$  do                                 $\triangleright r = \text{numero di blocchi in } \pi$ 
8:     if  $\pi_i \subseteq \mathcal{B}$  then
9:        $stmt_{int} := \text{CONVERT}(\pi, stmt)$ 
10:       $\mathcal{F} := \text{ASSIGNMENT}(\mathcal{P}'_i, stmt_{int})$                          $\triangleright \text{Poliedri non fattorizzati}$ 
11:       $\mathcal{O}.\text{add}(\mathcal{F})$ 
12:      if  $\mathcal{P}'_i := \emptyset$  then
13:        return  $\perp, \perp$ 
14:      else
15:         $\mathcal{O}.\text{add}(\mathcal{P}'_i)$ 
16:   return  $\mathcal{O}, \pi$ 

```

2.3.6 WIDENING (∇)

Per la doppia rappresentazione, l'operatore di *widening* necessita come parametri i generatori e i vincoli di \mathcal{P} e i vincoli di \mathcal{Q} . Il risultato dell'operazione $\mathcal{P} \nabla \mathcal{Q}$ conterrà i vincoli $\mathcal{C}_{\mathcal{Q}}$ che sono presenti in $\mathcal{C}_{\mathcal{P}}$ o che possono sostituirne un vincolo senza cambiare \mathcal{P} .

Per implementarlo è necessario effettuare una rifattorizzazione \mathcal{P}' del primo argomento \mathcal{P} con $\pi = \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$, mentre non è necessario rifattorizzare il secondo argomento. Successivamente, per ogni fattore \mathcal{Q}_i del secondo argomento si

individua il fattore corrispondente \mathcal{P}'_k in \mathcal{P}' e si procede a individuare l'insieme $\mathcal{C}_{\mathcal{O}_i}$ di vincoli di \mathcal{Q}_i che sono *stabili* rispetto a \mathcal{P}'_k : questi vincoli costituiscono il risultato del widening rispetto a quel fattore. Come caso speciale, se $\pi_{\mathcal{P}'_k} = \pi_{\mathcal{Q}_i}$, allora è possibile applicare direttamente l'operatore di widening per poliedri non fattorizzati.

Algoritmo 10 Widening

```

1: function WIDENIND( $\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{Q}, \pi_{\mathcal{Q}}$ )
2:    $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$ 
3:    $\mathcal{P}' := \text{REFACTOR}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$ 
4:    $\mathcal{O} := \emptyset$ 
5:   for each  $i$  in  $\{1, \dots, r\}$  do                                 $\triangleright q = \text{numero di blocchi in } \pi_{\mathcal{Q}}$ 
6:      $\mathcal{C}_{\mathcal{O}_i} := \emptyset$ 
7:      $k := j$ , t.c.  $\mathcal{X}_{\mathcal{Q}_i} \subseteq \mathcal{X}_j, \mathcal{X}_j \in \pi$ 
8:     if  $\pi_k = \pi_{\mathcal{Q}_i}$  then
9:        $\mathcal{Q}_i := \text{WIDENING}(\mathcal{P}'_k, \mathcal{Q}_i)$ 
10:    else
11:       $\mathcal{C}_{\mathcal{O}_i} := \text{SELECT\_STABLE\_CON}(\mathcal{C}_{\mathcal{Q}_i}, \mathcal{P}'_k)$ 
12:       $\mathcal{Q}_i := \text{NEW\_FACTOR}(\mathcal{C}_{\mathcal{O}_i})$ 
13:     $\mathcal{O}.\text{add}(\mathcal{O}_i)$ 
14:   return  $\mathcal{O}, \pi_{\mathcal{O}}$ 

```

3 IMPLEMENTAZIONE

Questo capitolo tratterà dell'implementazione effettiva dei poliedri fattorizzati e di alcune loro operazioni. La struttura è stata basata su di un *wrapper* preesistente nella *PPLite* mentre alcune delle implementazioni sono state attuate effettuando un porting dalla libreria *PPL*. La maggior parte delle operazioni sono state sviluppate come un wrapper per la *PPLite*, difatti le operazioni sui singoli poliedri sono gestite da essa. Lo scopo principale è quello di inserirsi all'interno dell'analisi statica e utilizzare i poliedri fattorizzati se e quando vengono generati poliedri che implicano un alto overhead per essere gestiti, in questo caso i suddetti poliedri vengono fattorizzati in maniera tale da diminuire il carico computazionale. L'implementazione è stata sviluppata interamente in *C++11* con l'obiettivo di utilizzare i metodi della *libreria standard* così da incrementare la leggibilità del codice. L'efficienza non è stata presa come obiettivo principale durante lo sviluppo, dando importanza alla correttezza e alla leggibilità del codice.

Durante l'implementazione è stata seguita una metodologia che è possibile suddividere in tre fasi:

1. inizialmente è stato effettuato un porting del codice dalla *PPL* alla *PPLite*, con adattamento alle strutture dati di quest'ultima;
2. successivamente è stato attuato un adeguamento incrementale agli standard di naming e codifica usati nella *PPLite*;
3. infine si è rivisto il codice scritto e si ci è apprestati a semplificarlo tramite l'utilizzo più sistematico di algoritmi della libreria standard e funzioni di supporto della *PPLite*. Questo ha permesso di avere un codice più corto e più facile da leggere.

3.1 F_Poly

La classe dove sono compresi gran parte dei metodi è `F_Poly`. È stata definita nel file `F_Poly.hh` dove sono presenti 92 metodi pubblici e 14 metodi privati.

Per la costruzione/distruzione degli oggetti è stata seguita la *Rule of Five*, quindi abbiamo:

- un costruttore esplicito, un costruttore per copia, un costruttore per spostamento;
- operazioni di assegnamento per copia e per spostamento;
- un distruttore.

In particolare, ci si è assicurati che il costruttore e l'assegnamento per spostamento, generati in automatico dal compilatore, fossero dichiarati `noexcept`: questo è richiesto allo scopo di evitare l'uso di copie costose quando si utilizzano alcuni contenitori e algoritmi della *STL*.

Abbiamo anche:

- un metodo `check_inv()` per controllare che l'invariante di classe non sia stata violata;
- le operazioni principali descritte nella sezione 2.3;
- funzioni di appoggio per gestire il poliedro.

Sono presenti delle strutture dati per la gestione dei fattori:

- `block` di tipo `Block`: un `std::vector` di `dim_type` che rappresenta un blocco \mathcal{X}_i che contiene alcune dimensioni del poliedro;
- `blocks` di tipo `Blocks`: un `std::vector` di `Block`, che esprime la partizione del poliedro;
- `factor` di tipo `Factor`: un fattore è un poliedro, difatti `Factor` è un alias per oggetti di tipo `Poly`;

- `factors` di tipo `Factors`: un `std::vector` di `Factor`, ognuno dei quali in corrispondenza posizionale con il corrispondente blocco in `blocks`.

Sono inoltre presenti tre variabili:

- `dim.type dim`: indicante le dimensioni del poliedro;
- `Topol topol`: indicante la topologia del poliedro;
- `bool is_empty`: indicante se il poliedro è vuoto o no.

3.1.1 CHECK_INV

Il metodo menzionato contiene tutti i controlli necessari per verificare che l'invariante di classe non sia stata violata, e che quindi il poliedro su cui si sta lavorando è ben formato. Le invarianti di classe che devono essere rispettate sono le seguenti:

- la dimensione del poliedro deve essere ≥ 0

```
if (dim < 0) {  
    reason = "F_Poly broken: invalid space dimension";  
    maybe_dump();  
    return false;  
}
```

;

- se il poliedro è vuoto, i fattori e i blocchi devono essere anch'essi vuoti

```
if (is_empty()) {  
    if (!(factors.empty() && blocks.empty())) {  
        reason = "F_Poly broken: empty polyhedron has factors"  
        ;  
        maybe_dump();  
        return false;  
    }  
    // No other check for an empty polyhedron.  
    return true;  
}
```

;

- il numero di blocchi deve essere uguale al numero di fattori

```
if (factors.size() != blocks.size()) {
    reason = "F_Poly broken: #factors != #blocks";
    maybe_dump();
    return false;
}
```

;

- la cardinalità totale dei blocchi deve essere uguale a dim

```
dim_type dim_ = 0;
for (const auto& block : blocks)
    dim_ += block.size();
if (dim != dim_) {
    reason = "F_Poly broken: space dimension mismatch";
    maybe_dump();
    return false;
}
```

;

- la dimensione di un fattore deve essere uguale alla dimensione del blocco che lo rappresenta

```
// Each factor space dim should match its block.
for (dim_type i = 0; i < num_rows(factors); ++i)
    if (factors[i].space_dim() != num_rows(blocks[i])) {
        reason = "F_Poly broken: factor vs block space dim mismatch";
        maybe_dump();
        return false;
    }
```

;

- nessun fattore deve essere vuoto

```
if (std::any_of(factors.begin(), factors.end(),
               std::mem_fn(&Poly::is_empty))) {
```

```

    reason = "F_Poly broken: empty factor";
    maybe_dump();
    return false;
}

```

```
;
```

- ogni dimensione dello spazio deve apparire una e una volta soltanto nei blocchi e non devono esserci blocchi vuoti (ovvero, `blocks` codifica solo una partizione)

```

std::vector<bool> dims(dim, false);
for (const auto& block : blocks) {
    if (block.size() == 0) {
        reason = "F_Poly broken: found empty block";
        maybe_dump();
        return false;
    }
    for (const auto d : block) {
        if (d < 0 || d >= dim) {
            reason = "F_Poly broken: block contains an illegal
                    space dim";
            maybe_dump();
            return false;
        }
        if (dims[d]) {
            reason = "F_Poly broken: repeated space dim in
                    blocks";
            maybe_dump();
            return false;
        }
        dims[d] = true;
    }
}

```

```
;
```

- ogni dimensione dello spazio deve essere presente in un blocco

```

if (std::find(dims.begin(), dims.end(), false) != dims.end
    ()) {

```

```

    reason = "F_Poly broken: a space dim is missing from
              blocks";
    maybe_dump();
    return false;
}

```

3.2 OPERAZIONI DI BASE

In questa sezione verranno descritte le funzione che più vengono utilizzate all'interno di `F_Poly.cc`:

```

static Factors refactor(const Factors& f,
                       const Blocks& b1, const Blocks& b2);
static Blocks least_upper_bound(const Blocks& b1, const Blocks&
                                b2);
static Blocks merge(const Blocks& b, const Block& added);

```

3.2.1 REFACTOR

La semantica rimane invariata rispetto a quanto descritto nell'Algoritmo 3. Per la *join* (\bowtie) tra i fattori è stata utilizzata la `concatenate_assign()` della PPlite; una funzione omonima, per quanto riguarda il join dei blocchi, è stata implementata utilizzando `std::vector::insert`. È importante notare che questa operazione è ammissibile in quanto non c'è alcuna possibilità di dimensioni doppie all'interno dei blocchi interessati.

```

F_Poly::Factors
F_Poly::refactor(const Factors& fs,
                 const Blocks& bs1, const Blocks& bs2) {
    Blocks bs(bs2.size());
    Factors res(bs.size(), Factor(0, Spec_Elem::UNIVERSE));

    for (dim_type i = 0; i != num_rows(bs1); ++i) {
        for (dim_type j = 0; j != num_rows(bs2); ++j) {

```



```
        if (detail::are_disjoint(bs1[i], bs2[j]))
            continue;
        res[j].concatenate_assign(fs[i]);
        concatenate_assign(bs[j], bs1[i]);
    }
}
```

Per rimappare le dimensioni in maniera corretta viene utilizzata la funzione `map_space_dims()`, che prende come argomento un oggetto di tipo `Dims` che viene riempito delle varie dimensioni dei blocchi che saranno rimappati in base all'indice in cui si trovano.

```
// Remap those factors res[i] s.t. bs[i] != bs2[i]
for (dim_type i = 0; i != num_rows(bs2); ++i) {
    const auto& b = bs[i];
    const auto& b2 = bs2[i];
    assert(b.size() == b2.size());
    if (b == b2)
        continue;
    Dims pf(b.size());
    for (auto j = 0; j != b.size(); ++j) {
        for (auto k = 0; k != b2.size(); ++k) {
            if (b[j] == b2[k]) {
                pf[j] = k;
                break;
            }
        }
    }
    res[i].map_space_dims(pf);
}
return res;
```

Il metodo `are_disjoint(const Block& b1, const Block& b2)` viene utilizzato per vedere se due blocchi hanno almeno una variabile in comune:

```
inline bool
are_disjoint(const Block& b1, const Block& b2) {
    return std::find_first_of(b1.begin(), b1.end(),
                              b2.begin(), b2.end()) == b1.end();
}
```

Utilizza `std::find_first_of` e ritorna `true` se non trova alcun elemento del primo insieme nel secondo.

3.2.2 LEAST_UPPER_BOUND

L'implementazione di questo metodo non è stata menzionata in [6] ma ne è stato solo descritto l'aspetto matematico. In termini più pratici questa operazione prende due parametri `const Blocks& b1`, `const Blocks& b2` e, inizialmente, crea un blocco `lub` che sarà composto dalle dimensioni in comune tra `b1` e `b2`:

```
F_Poly::Blocks
F_Poly::least_upper_bound(const Blocks& b1, const Blocks& b2) {
    Blocks lub; dim_type i = 0;
    bool first = true;

    for (const auto &b11 : b1)
        for (const auto &b12 : b2)
            if (!detail::are_disjoint(b11, b12)) {
                if (first) {
                    lub.push_back(b12);
                    first = false;
                }
                else {
                    lub[i] = block_union(lub[i], b12);
                }
                ++i;
                first = true;
            }
}
```

Successivamente si itera su `lub` per unire i blocchi che presentavano dimensioni comuni fino a costruire la partizione `lub`.

```
for (auto it1 = lub.begin(); it1 != lub.end(); ++it1)
    for (auto it2 = it1 + 1; it2 != lub.end(); ++it2)
        if (!detail::are_disjoint(*it1, *it2)) {
            *it1 = block_union(*it1, *it2);
            lub.erase(it2);
        }
```

```
--it2;  
}
```

Per unire i blocchi è stata utilizzata la funzione `block_union(const Block& b1, const Block& b2)` che permette di unire i blocchi ed evitare che si vengano a creare dimensioni doppie.

```
F_Poly::Block  
F_Poly::block_union(const Block& b1, const Block& b2) {  
    Block out(b1);  
    bool add = true;  
    for (const auto dim2 : b2) {  
        for (const auto dim1 : b1)  
            if (dim1 == dim2) {  
                add = false;  
                break;  
            }  
        if (add)  
            out.push_back(dim2);  
        else  
            add = true;  
    }  
    return out;  
}
```

3.2.3 MERGE

La funzione `merge` prende due parametri, un `Blocks b` e un `Block added` e genera un `Blocks out` che consiste nell'unione di `b` e `added`. La funzione cicla sul primo parametro, e da qui può essere suddivisa in tre parti principali:

- si controlla se i blocchi `b[i]` e `added` hanno variabili in comune, in caso positivo (se è la prima volta che si trovano blocchi non disgiunti) si aggiunge `b[i]` ad `out`:

```
Blocks out;  
bool first = true;  
dim_type index = 0;  
for (dim_type i = 0; i < num_rows(b); ++i) {
```

```
if (!detail::are_disjoint(b[i], added)) {
    if (first) {
        out.push_back(b[i]);
        first = false;
        index = i;
    }

```

;

- se i blocchi sono disgiunti ma non è la prima volta che se ne trovano:

```
else {
    concatenate_assign(out[index], b[i]);
}

```

;

- se invece i blocchi sono disgiunti:

```
else
    out.push_back(b[i]);
}

```

.

Infine si ritorna out.

3.3 OPERAZIONI SUI POLIEDRI

Qui verranno mostrate le implementazioni delle operazioni discusse nella Sezione 2.3.

3.3.1 INCLUSION

Come descritto nella Sezione 2.3.1, questa operazione è utile per controllare se un poliedro è incluso in un altro.

La funzione prende come parametri un insieme di fattori e i loro relativi blocchi. Si calcola il *lub* tra i blocchi passati come parametri e i blocchi di **this**, il risultato è utilizzato per rifattorizzare i fattori di **this** (in un nuovo insieme di fattori P) e i fattori passati come parametro (in un nuovo insieme

di fattori Q). Una volta fatto ciò non si fa altro che scorrere i vari fattori e richiamare la funzione `contains` della *PPLite*.

```
bool
F_Poly::inclusion(const Factors& f, const Blocks& b) const {
    Blocks b_lub = least_upper_bound(blocks, b);
    Factors P = refactor(factors, blocks, b_lub);
    Factors Q = refactor(f, b, b_lub);

    for (const auto& P_factor : P)
        for (const auto& Q_factor : Q)
            if (!P_factor.contains(Q_factor))
                return false;

    return true;
}
```

3.3.2 JOIN

Questa operazione permette di unire due poliedri. La funzione prende due parametri: un insieme di fattori `fs` e i suoi blocchi `bs`. La sua implementazione può essere suddivisa in tre fasi:

1. inizialmente vengono creati tutti li oggetti necessari:
 - `Pr` e `Qr` sono i due insiemi di fattori che conterranno i poliedri di `this` e di `fs`, entrambi rifattorizzati in base ai loro rispettivi blocchi e al *lub* tra `blocks` e `bs`;
 - `O` e `U` saranno, rispettivamente, i fattori e i blocchi finali (contenenti quindi il risultato della funzione);
 - `Pt`, `Qt` e `bpo` sono oggetti temporanei il cui scopo verrà spiegato successivamente.

```
Factors Pr;
Factors Qr;
Factors O = Factors();
Factor Pt;
Factor Qt;
```

```
Block bpo = Block();
Blocks U = Blocks();
Blocks lub = least_upper_bound(blocks, bs);
bool first = true;
Pr = refactor(factors, blocks, lub);
Qr = refactor(fs, bs, lub);
```

2. successivamente si procede alla costruzione dei vari poliedri che andranno a formare il join finale. Inizialmente si controlla, per ogni poliedro, se $Pr[i] = Qr[i]$, se sono uguali allora dentro O e U vengono inseriti rispettivamente $Pr[i]$ e il suo blocco:

```
for (dim_type i = 0; i < num_rows(Pr); ++i) {
    if (Pr[i] == Qr[i]) {
        O.push_back(Pr[i]);
        U.push_back(lub[i]);
    }
```

se non sono uguali ed è la prima volta che si verifica questa situazione, i blocchi $Qr[i]$ e $Rr[i]$ vengono inseriti rispettivamente negli oggetti temporanei bpo , Qt e Pt :

```
else {
    if (first) {
        bpo = lub[i];
        Qt = Qr[i];
        Pt = Pr[i];
        first = false;
```

se invece non è la prima volta, vengono semplicemente concatenati invece che aggiunti:

```
else {
    concatenate_assign(bpo, lub[i]);
    Pt.concatenate_assign(Pr[i]);
    Qt.concatenate_assign(Qr[i]);
}
```

3. a questo punto è necessario aggiungere a O e U i poliedri che sono risultati essere differenti nel punto precedente. Infine alle strutture in `this` si assegnano i poliedri e blocchi su cui è stato applicato il join:

```
Pt.poly_hull_assign(Qt);
U.push_back(bpo);
O.push_back(Pt);
factors = 0;
blocks = U;
dim = space_dim(blocks);
```

3.3.3 ADD_CON

Il metodo `add_con` non è altro che l'implementazione dell'Algoritmo 8. Per poter sviluppare questa operazione sono necessarie due diverse funzioni helper oltre a `refactor` e `merge`:

- `extract_block()`: una funzione che, preso un vincolo, restituisce il blocco contenente tutte le variabili utilizzate in esso:

```
inline Block
extract_block(const Linear_Expr& e) {
    Block var_block;
    for (dim_type i = 0; i != e.space_dim(); ++i)
        if (e.get(Var(i)) != 0)
            var_block.push_back(i);
    return var_block;
}
```

non facciamo altro che iterare una `Linear_Expr` inserendo tutte le variabili presenti in quest'ultima in un `std::vector`.

- `convert`: è una funzione che permette di mappare le dimensioni di un vincolo. Definendo come *esterne* le dimensioni del poliedro nella sua interezza e *interne* quelle di un suo fattore, questo metodo effettua la traduzione da vincolo esterno a vincolo interno in base al blocco che viene passato.

Inizialmente viene creato un blocco `b` che conterrà le variabili utilizzate nel vincolo `c` passato come parametro alla funzione. Successivamente possono presentarsi due casi nel caso in cui il `b` sia vuoto:

- se il vincolo è tautologico, quindi vero in ogni possibile interpretazione, si ritorna;
- se il vincolo è inconsistente, il poliedro viene settato a **empty** e si ritorna.

```
void
F_Poly::add_con(const Con& c) {
    if (is_empty())
        return;

    Block b = detail::extract_block(c);
    if (b.size() == 0) {
        if (c.is_tautological())
            return;
        else {
            assert(c.is_inconsistent());
            set_empty();
            return;
        }
    }
}
```

Se invece **b** non è vuoto si procede con l'aggiunta del vincolo al poliedro.

Inizialmente viene fatto un merge tra i blocchi del poliedro e il blocco **b** calcolato precedentemente, successivamente si rifattorizza in base al blocco risultante dal merge e, quest'ultimo, sostituirà i blocchi "vecchi" del poliedro.

Una volta fatto ciò, si cicla all'interno dei blocchi e, non appena si trova un blocco non disgiunto dal blocco **b**, si definisce la variabile **c_int** che conterrà il vincolo **c** ma con le variabili adattate a quelle del blocco **blocks[i]**, successivamente si aggiunge il vincolo utilizzando la funzione **add_con** della classe **Poly** e, se dopo aver aggiunto il vincolo il fattore risultasse vuoto, allora il poliedro viene settato a **empty**.

```
Blocks blocks_out = merge(blocks, b);
factors = refactor(factors, blocks, blocks_out);
blocks = blocks_out;

for (dim_type i = 0; i != num_rows(blocks); ++i) {
```



```

    if (detail::are_disjoint(b, blocks[i]))
        continue;
    Con c_int = detail::convert(c, blocks[i]);
    factors[i].add_con(std::move(c_int));
    if (factors[i].is_empty()) {
        set_empty();
        return;
    }
}
}
}

```

3.3.4 AFFINE_IMAGE E AFFINE_PREIMAGE

La funzione `affine_image` è l'implementazione dell'operazione *assignment* vista nell'Algoritmo 9. A differenza di quest'ultimo, per rimanere coerenti con le definizioni della PPLite, la funzione prende in input:

- Una variabile `var` di tipo `Var`;
- Un'espressione lineare `expr` di tipo `Linear_Expr`;
- Un intero `inhomo` di tipo `Integer` che rappresenta il *termine noto*;
- Un intero `den` di tipo `Integer` che rappresenta il *denominatore*.

Questi parametri formeranno lo statement $var = \frac{expr}{den} + inhomo$ equivalente a $x_i = ax + \epsilon$ dell'Algoritmo 9. Le uniche differenze con lo pseudocodice sono che è necessario convertire sia `var` che `expr` in base al blocco corrispondente. Una volta convertite non si fa altro che chiamare la `affine_image` della PPLite per ogni fattore.

```

void
F_Poly::affine_image(Var var, const Linear_Expr& expr,
                    const Integer& inhomo,
                    const Integer& den) {
    Block b = detail::extract_block(expr);
    detail::add_var(b, var);
    Blocks bs_out = merge(blocks, b);
}

```

```

factors = refactor(factors, blocks, bs_out);
blocks = bs_out;
for (dim_type i = 0; i < num_rows(factors); ++i) {
    if (detail::are_disjoint(b, blocks[i]))
        break;
    Var v_int = detail::convert(var, blocks[i]);
    Linear_Expr le_int = detail::convert(expr, blocks[i]);
    factors[i].affine_image(v_int, le_int, inhom, den);
    if (factors[i].is_empty())
        set_empty();
}
}

```

La funzione `affine_preimage` è identica a quella appena descritta, con l'unica differenza di chiamare la `affine_preimage` della PPLite per ogni fattore.

3.4 ALTRE OPERAZIONI

Oltre alle operazioni descritte in [6] ne sono state aggiunte altre per rendere più completa la classe. Tutte le operazioni descritte successivamente non sono altro che il corrispettivo sui poliedri fattorizzati di metodi omonimi presenti nella PPLite.

3.4.1 ADD_SPACE_DIMS E REMOVE_SPACE_DIMS

Funzioni che permettono l'aggiunta e la rimozione di dimensioni del poliedro. Ne sono state implementate tre versioni:

- `add_space_dims`: permette di aggiungere n (con n parametro della funzione) dimensioni al poliedro. Quest'ultimo viene incorporato nel nuovo spazio vettoriale. Avendo dei fattori implica che per ogni dimensione aggiunta deve essere aggiunto un nuovo blocco singoletto contenente la nuova dimensione e un rispettivo fattore *universo* di dimensione 1. È presente anche un parametro bool `project` (settato a `false` di default) che permette, se passato come `true`, di non incorporare il poliedro nel nuovo spazio vettoriale. Difatti, per ogni dimensione aggiunta viene creato un

blocco singoletto che contiene la dimensione `dim` del poliedro, mentre il fattore viene creato da un sistema di vincoli che indica che la sua unica dimensione è vincolata ad assumere il valore 0:

```
void
F_Poly::add_space_dims(dim_type m, bool project) {
    assert(m >= 0);
    if (m == 0)
        return;
    if (is_empty()) {
        dim += m;
        return;
    }
    Factor f(1, Spec_Elem::UNIVERSE, topology());
    if (project)
        f.add_con(Var(0) == 0);
    factors.insert(factors.end(), m, f);
    for (dim_type i = 0; i != m; ++i) {
        blocks.emplace_back(1, dim);
        ++dim;
    }
}
```

- `remove_space_dim`: permette di rimuovere una dimensione dal poliedro. Il problema principale è che, rimuovendo una dimensione, è necessario riorganizzare i blocchi in modo tale da essere uniformati all'insieme $\{1, \dots, dim\}$. Questa operazione è riservata a una funzione chiamata `reduce_blocks`.

La funzione inizia con un doppio ciclo `for` che permette di scorrere all'interno di ogni blocco del poliedro. Per ogni blocco, se il valore di `blocks[i][j]` è uguale a `v`, possiamo avere diverse casistiche:

- il blocco `blocks[i]` è formato da un singolo elemento: in questo caso viene eliminato tutto il blocco e il corrispondente fattore;
- il blocco `blocks[i]` è formato da più elementi: in questo caso viene eliminato solo il `j`-esimo elemento dell'`i`-esimo blocco, il fattore viene gestito tramite la `remove_space_dim` della classe `Poly`.

```
void
```

```

F_Poly::remove_space_dim(const dim_type v) {
    for (dim_type i = 0; i < num_rows(blocks); ++i)
        for (dim_type j = 0; j < num_rows(blocks[i]); ++j)
            if (v == blocks[i][j]) {
                if (num_rows(blocks[i]) == 1) {
                    blocks.erase(blocks.begin() + i);
                    factors.erase(factors.begin() + i);
                    reduce_blocks(blocks, v);
                }
                else {
                    blocks[i].erase(blocks[i].begin() + j);
                    factors[i].remove_space_dim(Var(v));
                    reduce_blocks(blocks, v);
                }
                --dim;
                return;
            }
}

```

È presente anche la versione `remove_space_dims(const Index_Set& vars)` che permette di rimuovere tutte le dimensioni presenti dentro `vars`.

- `remove_higher_space_dims`: è una variante della funzione precedente, rimuove le n dimensioni (con n parametro della funzione) più grandi del poliedro. È presente nel wrapper utilizzato in quanto risulta essere più efficiente nella classe `Poly`, per quanto riguarda `F_Poly` non risulta esserlo considerato che le dimensioni da rimuovere non sono necessariamente le ultime. Viene implementata quindi utilizzando la funzione descritta precedentemente.

```

void
F_Poly::remove_higher_space_dims(dim_type new_dim) {
    if (is_empty()) {
        dim = new_dim;
        return;
    }
    for (dim_type i = space_dim(); i-- > new_dim; )
        remove_space_dim(i);
}

```

3.4.2 ADD_GEN

Il metodo `add_gen` viene utilizzato per aggiungere un generatore al poliedro. Per ogni generatore che viene passato come parametro possono esserci tre casi:

- **Il poliedro è vuoto:** in questo caso si controlla che il generatore sia un punto, successivamente tutto il poliedro viene settato come *universo* e, per ogni fattore, viene aggiunto un vincolo $d \cdot \text{Var}(0) = \text{coeff}(\text{Var}(i))$

```
if (is_empty()) {
    assert(g.is_point());
    *this = F_Poly(dim, Spec_Elem::UNIVERSE, topology());
    for (dim_type i = dim; i-- > 0; )
        factors[i].add_con(g.divisor() * Var(0) == g.coeff(Var(
            i)));
    assert(check_inv());
    return;
}
```

- **Il poliedro non è vuoto e il generatore è un punto:** in questo caso non si fa altro che costruire un insieme di fattori, ognuno dei quali conterrà il generatore passato come parametro; infine questi fattori vengono aggiunti al poliedro tramite la *join*:

```
if (g.is_point()) {
    Factors fs_g;
    const auto nb = num_rows(blocks);
    fs_g.reserve(nb);
    for (dim_type i = 0; i != nb; ++i) {
        const auto& bi = blocks[i];
        const auto nbi = num_rows(bi);
        fs_g.emplace_back(nbi, Spec_Elem::UNIVERSE, topology(
        ));
        auto& fi = fs_g.back();
        for (dim_type j = 0; j != nbi; ++j)
            fi.add_con(g.divisor() * Var(j) == g.coeff(Var(bi[j
            ])));
    }
    join(fs_g, blocks);
    assert(check_inv());
    return;
}
```

```
}

```

- **Il poliedro non è vuoto e il generatore non è un punto:** in questo caso viene creato un blocco \mathcal{B} contenente tutte le dimensioni che appaiono nel generatore, successivamente si crea un blocco $\pi = \pi_{\mathcal{P}} \uparrow \mathcal{B}$ tramite il quale si ri-fattorizza il nostro poliedro in \mathcal{P}' . Infine, per ogni blocco che compare anche nel generatore si richiama la `add_gen()` della PPLite

```
assert(!is_empty() && !g.is_point());
Block b_g = detail::extract_block(g);
Blocks b_out = merge(blocks, b_g);
factors = refactor(factors, blocks, b_out);
blocks = b_out;

for (dim_type i = 0; i != num_rows(blocks); ++i) {
    if (detail::are_disjoint(b_g, blocks[i]))
        continue;
    auto g_int = detail::convert(g, blocks[i]);
    factors[i].add_gen(std::move(g_int));
    break;
}
assert(check_inv());

```

3.4.3 PREDICATI

Sono presenti diversi predicati in `F_Poly.hh` che permettono di controllare lo stato del poliedro, i principali sono:

- `bool is_empty` non fa altro che dire se il poliedro è vuoto o no;
- `bool is_minimized` controlla che tutti i fattori siano minimizzati, per farlo viene utilizzato `std::all_of` passando come predicato unario il puntatore alla funzione `is_minimized` della classe `Poly`:

```
bool is_minimized() const {
    return is_empty()
        || std::all_of(factors.begin(), factors.end(),
                      std::mem_fn(&Poly::is_minimized));
}

```

- `bool is_necessarily_closed` dice se la topologia del poliedro è chiusa o no;
- `bool is_universe` dice se il poliedro è di tipo universo;
- `bool is_bounded` controlla se il sistema di equazioni che definisce il poliedro è bounded
- `bool equals(const F_Poly& y)` controlla se `this` è uguale a `y`. Inizialmente viene calcolato il *lub* tra i blocchi di `this` e `y`, successivamente vengono rifattorizzati i fattori in base al lub calcolato e vengono assegnati a `fx` e `fy`, infine si controlla se ogni fattore in `fx` è uguale a ogni fattore in `fy`:

```
bool
F_Poly::equals(const F_Poly& y) const {
    if (dim != y.space_dim())
        return false;
    if (is_empty() && y.is_empty())
        return true;

    Blocks b_lub = least_upper_bound(blocks, y.blocks);
    Factors fx = refactor(factors, blocks, b_lub);
    Factors fy = refactor(y.factors, y.blocks, b_lub);

    for (auto i = fx.size(); i-- > 0; )
        if (fx[i] != fy[i])
            return false;

    return true;
}
```

3.4.4 OVERLOAD DEGLI OPERATORI

È stato anche fatto un overload negli operatori `=` e `!=`, l'implementazione è semplice e si basa sui predicati descritti nella sezione precedente:

```
inline bool
operator==(const F_Poly& x, const F_Poly& y) {
```

```
    return x.equals(y); }

inline bool
operator!=(const F_Poly& x, const F_Poly& y) {
    return !(x == y); }
```


4 VALUTAZIONE SPERIMENTALE

In questa sezione verranno mostrati i diversi test di correttezza che sono stati effettuati e un test di efficienza per palesare l'effettivo guadagno che ci si aspettava.

Per quanto riguarda i test di correttezza, sono stati utilizzati quelli già presenti nella PPLite. Per farlo, è stato necessario fare in modo che, ogni qualvolta viene costruito un oggetto di tipo `Poly` ne viene costruito, in realtà, uno di tipo `F_Poly`. Per l'esattezza, sono stati completati con successo 143 test di correttezza, tra cui:

- **addcons1.cc, addcons2.cc, addcons3.cc**: 23 test sull'aggiunta di vincoli;
- **addspacedims.cc**: 7 test sull'aggiunta di dimensioni;
- **affinedim1.cc**: 4 test sulle dimensioni affini;
- **affineimage.cc**: 13 test sulle immagini affini;
- **affinepreimage.cc**: 16 test sulle preimmagini affini;
- **bounded.cc**: 5 test per verificare che il poliedro sia bounded;
- **constrains.cc**: 7 test sul metodo `constrains`;
- **disjoint.cc**: 5 test sulla correttezza del metodo `is_disjoint_from`;
- **empty1.cc**: 7 test sulla generazione di poliedri vuoti;
- **foldspacedims.cc**: 9 test sulla fold delle dimensioni;
- **hyper.cc**: 1 test sulla generazione di ipercubi;
- **hyperphull.cc**: 1 test sulla join di ipercubi;

- **inters.cc**: 13 test sull'intersezione di poliedri;
- **polyhull.cc**: 12 test sul join di poliedri;
- **randphull.cc**: 2 test randomizzati sul join di poliedri;
- **removehigherdims.cc**, **removespacedims.cc**: 18 test sulla rimozione di dimensioni.

Il test di efficienza, invece, si è basato su un esempio concreto preso da uno dei benchmark di *PHAVerLite*. Sono stati generati due poliedri di tipo **Poly** di dimensioni elevate, successivamente è stato applicato il join (\bowtie) tra questi due, in seguito sono stati costruiti due oggetti di tipo **F_Poly** passando ai costruttori i due poliedri di cui sopra e applicato il join, infine è stata applicata una funzione (resa disponibile dagli sviluppatori della *PPLite*) per convertire i poliedri fattorizzati in oggetti di tipo **Poly**.

I risultati sono stati questi:

5 CONCLUSIONE

In questa tesi si è quindi parlato della fattorizzazione dei poliedri, ovvero la tecnica di suddividere un poliedro in poliedri più piccoli, riducendo così il costo computazionale in tempo e memoria. Questo è reso possibile dal fatto che le variabili all'interno di un programma spesso non sono tutte legate tra loro e sono quindi suddivisibili in gruppo indipendenti dagli altri. Lo scopo è quindi quello di manipolare queste partizioni in maniera tale da renderle più piccole possibili.

All'interno della PPLite la fattorizzazione cartesiana non sarà utilizzata come sostituto della classe `Poly` bensì, verrà utilizzata laddove è computazionalmente conveniente farlo.

Per gli sviluppi futuri ci si concentrerà nel terminare le operazioni mancanti in modo tale da rendere completa la classe. Inoltre, in quanto non ci si è concentrati principalmente nell'efficienza del codice, sarà necessario rivederlo andando alla ricerca di punti critici per ottimizzare il codice e sfruttare al meglio le funzionalità offerte dal C++11.

BIBLIOGRAFIA

- [1] D. Arioli. Un wrapper per la fattorizzazione cartesiana dei poliedri convessi. Undergraduate thesis, Department of Mathematical, Physical and Computer Sciences, University of Parma, Italy, December 2017. In Italiano.
- [2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
- [3] A. Becchi. Poliedri NNC: una nuova rappresentazione e algoritmo di conversione. Undergraduate thesis, Department of Mathematical, Physical and Computer Sciences, University of Parma, Italy, September 2017. In Italiano.
- [4] A. Becchi and E. Zaffanella. An efficient abstract domain for not necessarily closed polyhedra. In A. Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 146–165. Springer, 2018.
- [5] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006.
- [6] G. Singh, M. Püschel, and M. T. Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 46–59, 2017.