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

Relatore:

Prof. Enea Zaffanella

Candidato:

Luigi Zaccone

ANNO ACCADEMICO 2017-2018

Indice

1 Introduzione

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 i semispazi e gli iperpiani del poliedro [1];

- 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 [1]. I generatori possono essere formati da:
  - **punto**: ogni elementi 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 $\boldsymbol{r} \in \mathbb{R}^n$ non nullo tale che per ogni $\boldsymbol{x} \in \mathcal{P}$ e per ogni $\mu \geq 0$ vale

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

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

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

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

    Se $\mathcal{P} = \varnothing$ 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 $0 \notin R$ e $0 \notin L$, 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 combinazioni convesse dei punti in P e come somma 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 $H_n = \{0 \leq x_i \leq 1 \mid i = 1, ..., n\}$ 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 [2].

2.1 Blocchi e fattori

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

Sia $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ 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 $\pi = \pi_P = \{\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_t\}$.

Esempio 2.1.1. Consideriamo

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

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:

$$\mathcal{C}_{P_1} = \{-x_1 \leq -1, x_1 \leq 4\} \qquad \mathcal{C}_{P_2} = \{-x_2 \leq -2, x_2 \leq 4\}$$
$$\mathcal{G}_{P_1} = \{L, R, (1,4)^T\} \qquad \mathcal{G}_{P_2} = \{L, R, (2,4)^T\}$$

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* $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. 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 πˆ' o, equivalentemente che πˆ' 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 $\bot = \{\{x_1\}, \{x_2\}, ..., \{x_n\}\}$.

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 $[0, ..., n]$ 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 $\pi_{\mathcal{P}}$ e $\pi_{\mathcal{Q}}$ 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

function LEAST-UPPER-BOUND($\pi_{\mathcal{P}}, \pi_{\mathcal{Q}}$)
    $\pi := \varnothing$
    for each $\mathcal{X}_{\mathcal{P}i}$ **in** $\pi_{\mathcal{P}}$ **do**
        $\mathcal{B} := \varnothing$
        for each $\mathcal{X}_{\mathcal{Q}k}$ **in** $\pi_{\mathcal{Q}}$ **do**
            if $\mathcal{X}_{\mathcal{P}i} \cap \mathcal{X}_{\mathcal{Q}k} \neq \varnothing$ **then**
                $\mathcal{B} := \mathcal{B} \cup \mathcal{X}_{\mathcal{Q}k}$
        π.add($\mathcal{B}$)
    while $\exists\ \pi_i, \pi_j \in \pi,\ i \neq j$ **t.c.** $\mathcal{X}_i \cap \mathcal{X}_j \neq \varnothing$ **do**
        $\pi := (\pi \setminus \{\pi_i, \pi_j\}) \cup \{\pi_i \cup \pi_j\}$
    return π

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

function MERGE($\pi, \mathcal{A}$)
    $\pi_m := \varnothing$
    $\mathcal{B} := \varnothing$
    for each $\mathcal{X}_i$ **in** π **do**
        if $\mathcal{X}_i \cap \mathcal{A} \neq \varnothing$ **then**
            $\mathcal{B} := \mathcal{B} \cup \mathcal{X}_i$
        else
            π_m.add($\mathcal{X}_i$)
    π_m.add($\mathcal{B}$)
    return π_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 $X = \{x_1, x_2, ..., x_n\}$ e presi $\pi_{\mathcal{P}} = \{X_{\mathcal{P}_1}, X_{\mathcal{P}_2}, ..., X_{\mathcal{P}_r}\}$, $\pi_{\mathcal{Q}} = \{X_{\mathcal{Q}_1}, X_{\mathcal{Q}_2}, ..., X_{\mathcal{Q}_s}\}$ 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

function REFACTOR($\mathcal{P}, \pi_{\mathcal{P}}, \pi$)
    $\mathcal{O}, \pi_{\mathcal{O}} := \varnothing$
    for each i **in** $\{1, ..., r\}$ **do** ▷ r = numero blocchi di $\pi_{\mathcal{P}}$
        $\mathcal{O}_i, \pi_{\mathcal{O}i} := \varnothing$
        for each j **in** $\{1, ..., m\}$ **do** ▷ m = numero blocchi di π
            if $\pi_{\mathcal{P}i} \cap \pi_i \neq \varnothing$ **then**
                $\mathcal{O}_i := \mathcal{O}_j \bowtie \mathcal{P}_i$
                $\pi_{\mathcal{O}i} := \pi_{\mathcal{O}i} \bowtie \pi_{\mathcal{P}i}$
        $\mathcal{O}$.add($\mathcal{O}_i$)
        $\pi_{\mathcal{O}}$.add($\pi_{\mathcal{O}i}$)
    REMAP-DIMENSIONS($\mathcal{O}, \pi_{\mathcal{O}}, \pi$)
    return $\mathcal{O}, \pi_{\mathcal{O}}$

Come spiegato precedentemente, la refactor prende in input un poliedro fattorizzato $\mathcal{P}$ e la sua partizione $\pi_{\mathcal{P}}$ per rifattorizzarlo in base al terzo parametro π. 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 π.

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 e, 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

```
function INCLUSION(𝒫, π_𝒫, 𝒬, π_𝒬)
    π := LEAS-UPPER-BOUND(π_𝒫, π_𝒬)
    𝒫' := REFACTOR(𝒫, π_𝒫, π)
    𝒬' := REFACTOR(𝒬, π_𝒬, π)
    for each k in {1, ..., r} do                ▷ r = numero di blocchi in π
        if 𝒫'_k non include 𝒬'_k then
            return False
    return True
```

2.3.2 Join ($\sqcup$)

Tramite la doppia reppresentazione, 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

```
function JOIN(P, π_P, O, π_O)
    if IS_EMPTY(P) then
        O, π_O := Q, π_Q
    else if IS_EMPTY(Q) then
        O, π_O := P, π_P
    else
        O, π_O := JOIN-POLY(P, π_P, O, π_O)
    return O, π_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 unito a un fattore comune $(\mathcal{P}'_i, \mathcal{Q}'_i)$ 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 $\mathcal{Q} \cup \mathcal{P} = \mathcal{Q}$ e anche $\mathcal{P} \cup \mathcal{Q} = \mathcal{Q}$.

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

function $\textsc{Join-Poly}(\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{O}, \pi_{\mathcal{O}})$
  $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$
  $\mathcal{P}' := \textsc{Refactor}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$
  $\mathcal{Q}' := \textsc{Refactor}(\mathcal{Q}, \pi_{\mathcal{Q}}, \pi)$
  for each i **in** $\{0, ..., |\mathcal{P}'|\}$ **do**
    if $\mathcal{P}'_i = \mathcal{Q}'_i$ **then**
      $\mathcal{O}.\mathrm{add}(\mathcal{P}'_i)$
      $\pi_{\mathcal{O}}.\mathrm{add}(\mathcal{X}'_i)$
    else
      $\mathcal{X}_{\mathcal{T}} := \mathcal{X}_{\mathcal{T}} \cup \pi_i$
      $\mathcal{P}_{\mathcal{T}} := \mathcal{P}_{\mathcal{T}} \bowtie \mathcal{P}'_i$
      $\mathcal{Q}_{\mathcal{T}} := \mathcal{Q}_{\mathcal{T}} \bowtie \mathcal{Q}'_i$
  $\mathcal{P}_{\mathcal{T}} := \mathcal{P}_{\mathcal{T}} \sqcap \mathcal{Q}_{\mathcal{T}}$
  $\pi_{\mathcal{O}}.\mathrm{add}(\mathcal{X}_{\mathcal{T}})$
  $\mathcal{O}.\mathrm{add}(\mathcal{P}_{\mathcal{T}})$
  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 $\mathcal{P} \sqcap \mathcal{Q} = \bot$ e $\mathcal{G}_{\mathcal{P}\sqcap\mathcal{Q}} = \varnothing$.

Per quanto riguarda i poliedri fattorizzati è necessario generare $\pi = \pi_{\mathcal{P}} \sqcap \pi_{\mathcal{Q}}$ 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* sarebbe un poliedro $\bot$. Se al contrario non fosse vuoto, aggiungo $\mathcal{F}$ a $\mathcal{O}$ per poi riprender 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. $\pi_{\mathcal{P}\sqcap\mathcal{Q}} = \pi_{\mathcal{P}} \sqcap \pi_{\mathcal{Q}}$ è una partizione ammissibile se $\mathcal{P} \sqcap \mathcal{Q} \neq \bot$, altrimenti $\bot$ è ammissibile.

Algoritmo 7 Meet

function $\textsc{Meet}(\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{Q}, \pi_{\mathcal{Q}})$
    $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$
    $\mathcal{P}' := \textsc{Refactor}(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$
    $\mathcal{Q}' := \textsc{Refactor}(\mathcal{Q}, \pi_{\mathcal{Q}}, \pi)$
    $\mathcal{O} := \varnothing$
    for each i **in** $\{1, ..., r\}$ **do** ▷ r = numero di blocchi in π
        if $\mathcal{P}'_i = \mathcal{Q}'_i$ **then**
            $\mathcal{O}.\mathrm{add}(\mathcal{P}'_i)$
        else
            $\mathcal{F} := \mathcal{P}'_i \bowtie \mathcal{Q}'_i$
            if $\textsc{is\_empty}(\mathcal{F})$ **then**
                return $\bot, \bot$
            $\mathcal{O}.\mathrm{add}(\mathcal{F})$
    $\pi_{\mathcal{O}} := \pi$
    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 necessare 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 in $\bot$ vengono modificati e il poliedro si rende *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 $\mathcal{O} \neq \bot$, altrimenti $\pi_{\mathcal{O}} = \bot$.

Algoritmo 8 Conditional

function CONDITIONAL(con)
  $\mathcal{B}$:= EXTRACT_BLOCK($\mathcal{P}$, $\pi_{\mathcal{P}}$, con)
  $\pi := \pi_{\mathcal{P}} \uparrow \mathcal{B}$
  $\mathcal{P}'$:= REFACTOR($\mathcal{P}, \pi_{\mathcal{P}}, \pi$)
  for each i **in** $\{1, ..., r\}$ **do** ▷ r = numero di blocchi in π
    if $\mathcal{B} \cap \pi_i \neq \varnothing$ **then**
      con_{int} := CONVERT_CON(π_i, con)
      $\mathcal{F}$:= ADD_CON($\mathcal{P}'_i$, con_{int})
      $\mathcal{O}$.add($\mathcal{F}$)
      if IS_EMPTY($\mathcal{O}_{\rangle}$) **then**
        return $\bot, \bot$
      $\mathcal{O}$.add($\mathcal{P}'_{\rangle}$)
  $\pi_{\mathcal{O}} := \pi$
  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

function $\textsc{Assignment}(\mathcal{P},\ \pi_{\mathcal{P}}, stmt)$
  let $stmt = (x_i = ax + \epsilon)$
  $\mathcal{B} :=$ EXTRACT_BLOCK$(stmt)$
  $\pi := \pi_{\mathcal{P}} \uparrow \mathcal{B}$
  $\mathcal{P}' :=$ REFACTOR$(\mathcal{P}, \pi_{\mathcal{P}}, \pi)$
  for each i **in** $\{1, ..., r\}$ **do** ▷ $r = |\mathcal{P}|$
    if $\pi_i \subseteq \mathcal{B}$ **then**
      $stmt_{int} :=$ CONVERT$(\pi, stmt)$
      $\mathcal{F} :=$ ASSIGNMENT$(\mathcal{P}'_i, stmt_{int})$ ▷ Poliedri non fattorizzati
      $\mathcal{O}$.add$(\mathcal{F})$
      if $\mathcal{P}'_i := \varnothing$ **then**
        return $\bot, \bot$
    else
      $\mathcal{O}$.add$(\mathcal{P}'_i)$
  return $\mathcal{O}, \pi$

2.3.6 Widening (∇)

Per la doppia rappresentazione, l'operatore di *widening* necessità come parametri i generatori e i vincoli di $\mathcal{P}$ e i vincolo 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 uan 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. È possibile, come caso speciale, che se $\pi_{\mathcal{P}'_k} = \pi_{\mathcal{Q}_i}$, allora è possibile applicare direttamente l'operatore di widening per poliedri non fattorizzati.

Algoritmo 10 Meet

function WIDENIND($\mathcal{P}, \pi_{\mathcal{P}}, \mathcal{Q}, \pi_{\mathcal{Q}}$)
  $\pi := \pi_{\mathcal{P}} \sqcup \pi_{\mathcal{Q}}$
  $\mathcal{P}' :=$ REFACTOR($\mathcal{P}, \pi_{\mathcal{P}}, \pi$)
  $\mathcal{O} := \varnothing$
  for each i **in** $\{1, ..., r\}$ **do** ▷ q = numero di blocchi in $\pi_{\mathcal{Q}}$
    $\mathcal{C}_{\mathcal{O}_i} := \varnothing$
    $k := j$, t.c. $\mathcal{X}_{\mathcal{Q}_i} \subseteq \mathcal{X}_j, \mathcal{X}_j \in \pi$
    if $\pi_k = \pi_{\mathcal{Q}_i}$ **then**
      $\mathcal{Q}_i :=$ WIDENING($\mathcal{P}'_k, \mathcal{Q}_i$)
    else
      $\mathcal{C}_{\mathcal{O}_i} :=$ SELECT_STABLE_CON($\mathcal{C}_{\mathcal{Q}_i}, \mathcal{P}'_k$)
      $\mathcal{Q}_i :=$ NEW_FACTOR($\mathcal{C}_{\mathcal{O}_i}$)
    $\mathcal{O}$.add($\mathcal{O}_i$)
  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.

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 stutture 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 Operazione di base

Nel seguito di descrivono le implementazioni delle operazioni di base.

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.

Diverse sono le funzioni più significative definite all'interno di `F_Poly.hh`:

```
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);
void join(const Factors& f, const Blocks& b);
void join(const F_Poly& f);
void affine_image(Var var, const Linear_Expr& expr,
                  const Integer& inhomo = Integer::zero(),
                  const Integer& den = Integer::one());


void affine_preimage(Var var, const Linear_Expr& expr,
                     const Integer& inhomo = Integer::zero(),
                     const Integer& den = Integer::one());
void add_con(const Con& c);
void add_gen(const Gen& c);
bool inclusion(const Factors& f, const Blocks& b) const;
```

3.2.1 REFACTOR

La semantica rimane invariata 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::insert`[1]. E importante notare che questa operazione sia ammissibile in quanto non c'è alcuna possibilità di dimensioni doppie all'interno dei blocchi interessati.

```
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 = b.size(); j-- > 0; )
      pf[b[j]] = b2[j];
    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 sono uguali.

[1] `template <class InputIt> iterator insert(const_iterator pos, InputIt first, InputIt last);`

```
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`[2] e ritorna falso se trova un elemento che è presente in un insieme e non in un altro.

3.2.2 LEAST_UPPER_BOUND

L'implementazione di questo metodo non è stata menzionata in [2] 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 non in comune tra `b1` e `b2`:

```
Blocks lub; dim_type i = 0;
bool first = true;

for (const auto &bl1 : b1)
  for (const auto &bl2 : b2)
    if (!detail::are_disjoint(bl1, bl2)) {
      if (first) {
        lub.push_back(bl2);
        first = false;
      }
      else {
        lub[i] = block_union(lub[i], bl2);
      }
  ++i;
  first = true;
}
```

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

[2] `template <class InputIt, class ForwardIt> InputIt find_first_of(InputIt first, InputIt last, ForwardIt s_first, ForwardIt s_last);`

```
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 al 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 ADD_CON

Il metodo `add_con` non è altro che l'implementazione dell'Algoritmo 8. Per poter sviluppare questa operazione sono necessare 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.

3.2.4 AFFINE_IMAGE E AFFINE_PREIMAGE

Questa funzione 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 l'*inhomogeneus term*;
- 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, inhomo, 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.

3.3 Altre operazioni

Oltre alle operazioni descritte in [2] 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.

3.3.1 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 $\{1, ..., dim\}$. Questa operazione è riservata a una funzione chiamata `reduce_blocks`:

```
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`: la sua implementazione utilizza la funzione precedente e non fa altro che rimuovere le n dimensioni (con n parametro della funzione) più grandi del poliedro:

```
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);
}
```

3.3.2 ADD_CON E ADD_GEN

Le due funzioni vengono utilizzate rispettivamente per aggiungere un vincolo e un generatore al poliedro. Per quanto riguarda la prima, non è altro che la *conditional* vista nell'Algoritmo 8, la seconda è un caso a parte.

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 $d \cdot 0 = coeff(Var(i))$

```
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 $\mathcal{B}$ contentente 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());
```

4 Conclusione

Bibliografia

[1] Anna Becchi. Poliedri nnc: una nuova rappresentazione e algoritmo di conversione. Technical report, 2017.

[2] Martin Vechev Gagandeep Singh, Markus Püschel. Fast polyhedra abstract domain. Technical report, 2017.