Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

30 settembre 2019

Sommario

L'algoritmo di Cooper permette di effettuare l'eliminazione dei quantificatori universali da formule dell'aritmetica di Presburger. In questo documento verrà descritto l'algoritmo e verrà discussa una semplice implementazione in C di una versione ridotta dell'algoritmo atta ad interfacciarsi al software di model checking MCMT.[1]

1 Aritmetica di Presburger

Sia $\mathbb{Z}$ l'anello degli interi, sia $\Sigma_{\mathbb{Z}}$ la segnatura $\{0, +, -, <\}$ e sia $\mathcal{A}_{\mathbb{Z}}$ il modello standard degli interi. Definiamo la teoria dell'**aritmetica di Presburger** come l'insieme $T_{\mathbb{Z}} = Th(\mathcal{A}_{\mathbb{Z}}) = Th(\mathbb{Z}, 0, 1, +, -, <)$ di tutte le $\Sigma_{\mathbb{Z}}$-formule vere in $\mathcal{A}_{\mathbb{Z}}$. Tale teoria non ammette l'eliminazione dei quantificatori.

Consideriamo ora la segnatura estesa $\Sigma^*_{\mathbb{Z}}$ ottenuta aggiungendo a $\Sigma_{\mathbb{Z}}$ un'infinità di predicati unari di divisibilità D_k per ogni $k \geq 2$, dove $D_k(x)$ indica che $x \equiv_k 0$. Sia $T^*_{\mathbb{Z}}$ l'insieme delle $\Sigma_{\mathbb{Z}}$-formule vere nell'espansione $\mathcal{A}^*_{\mathbb{Z}}$ ottenuta da $\mathcal{A}_{\mathbb{Z}}$.

Nel 1930 Mojżesz Presburger ha esibito un algoritmo di eliminazione dei quantificatori[2] per $T^*_{\mathbb{Z}}$ e nel 1972 Cooper ha fornito una versione migliorata basata sull'eliminazione dei quantificatori da formule nella forma $\exists\, x \,.\, \varphi$, dove φ è una formula senza quantificatori arbitraria.

1.1 Formalizzazione dell'aritmetica di Presburger

Si definiscano i simboli dell'aritmetica di Presburger:

$$\mathscr{L} = \{(,), \wedge, \vee, \neg, \exists, \forall, =, <, |, +, -, 0, 1, x, y, z, \dots\}$$

I simboli $x, y, z, \dots$ sono chiamati variabili, essi possono ammettere un pedice. Una espressione è una successione finita di simboli, si chiami quindi $\mathscr{L}^+$ il linguaggio delle espressioni nell'aritmetica di Presburger. Un termine è definito nel modo seguente:

- Le variabili e i simboli 0 e 1 sono termini.
- Se t_1 e t_2 sono termini, lo sono anche $(t_1 + t_2)$ e $-t$.
- Questi sono gli unici termini.

Una formula atomica è una espressione del tipo $t_1 < t_2$, $k \mid t_1$ o $t_1 = t_2$, dove t_1 e t_2 sono termini e $k \in \mathbb{N} \smallsetminus \{0, 1\}$.

Una formula è definita come segue:

[1] silvio ghilardi. *mcmt: model checker modulo theories*. `http://users.mat.unimi.it/users/ghilardi/mcmt/`. 2018.

[2] Mojżesz Presburger. "On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation". In: *Hist. Philos. Logic* 12.2 (1991). Translated from the German and with commentaries by Dale Jacquette, pp. 225–233. ISSN: 0144-5340. DOI: `10.1080/01445340910883718 7`.

- Una formula atomica è una formula
- Se A e B sono formule e x è una variabile, allora $\exists\, x\, A$, $\forall\, x\, B$, $(A \wedge B)$, $(A \vee B)$ e $\neg A$ sono ancora formule.
- Queste sono le sole formule.

Si chiami frase una formula che non ha variabili libere. La semantica del linguaggio è quella naturale, si osservi solo che, per convenienza di scrittura, verranno usati anche i numerali $(2, 3, \dots)$ definiti come somme iterate:

$$n \equiv \underbrace{1 + 1 + \cdots + 1}_{n}$$

Inoltre potranno venire usati, sempre solamente per convenienza di scrittura, anche i seguenti altri simboli:

- $t_1 \leq t_2$ al posto di $t_1 = t_2 \vee t_1 < t_2$.
- $t_1 > t_2$ al posto di $\neg\, t_1 \leq t_2$.
- $t_1 \geq t_2$ al posto di $t_1 = t_2 \vee t_1 > t_2$.
- $k \nmid t_1$ al posto di $\neg\, k \mid t_1$.

Dove t_1 e t_2 sono termini e $k \in \mathbb{N} \smallsetminus \{0, 1\}$. Tali simboli non sono da considerarsi parte dell'aritmetica di Presburger, ma solo come notazione per abbreviare e rendere maggiormente suggestive le prossime affermazioni.

2 L'algoritmo di Cooper

L'algoritmo di Cooper prevede come un input una formula in $\mathscr{L}^+$ e una variabile, per poi fornire in output una frase equivalente priva di tale variabile. Si osservi che se esso viene iterato sulle diverse variabili presenti nella formula in input permette di giungere ad una frase (i.e. una formula senza variabili libere). A questo punto è sufficiente valutare la frase per determinare la verità della formula in ingresso, in questo senso si può interpretare l'algoritmo di Cooper come procedura decisionale. Vengono quindi mostrati i passaggi effettuati dall'algoritmo in una singola iterazione.

Consideriamo una formula in ingresso della forma $\exists\, x\, F(x)$, dove F è senza quantificatori. Innanzitutto si osservi che assumere il quantificatore esistenziale non è limitativo in quanto se fosse presente $\forall\, x$, esso potrebbe essere semplicemente sostituito con $\neg\, \exists\, x\, \neg$.

Step 1. Si eliminano le negazioni logiche portando i $\neg$ il più lontano possibile dagli atomi (per esempio usando le leggi di De Morgan) e successivamente si sostituiscono i letterali che consistono di atomi negati con atomi equivalenti non negati. (e.g. sostituire $\neg(x \leq a)$ con $x > a$) A questo punto si sostituiscono tutte le formule che contengono altri simboli relazionali che non siano $<$, $\mid$ or $\nmid$ in formule equivalenti contenenti solo $<$.

Step 2. Sia δ' il minimo comune multiplo dei coefficienti della x, si moltiplicano ambo i lati di tutti gli atomi contenenti x per costanti appropriate in modo tutti i coefficienti della x siano δ'. Infine si sostutuisce $\exists\, x\, F(\delta' x)$ con $\exists\, x\, (F(x) \wedge \delta' \mid x)$. Si ha quindi ottenuto una formula equivalente dove ogni atomo che non contiene la x deve essere obbligatoriamente in una delle seguenti forme.

A. $x < a_i$

B. $b_i < x$

C. $\delta_i \mid x + c_i$

D. $\epsilon_i \nmid x + d_i$

Dove a_i, b_i, c_i e d_i sono espressioni senza x e δ_i e ϵ_i sono interi positivi.

Step 3. Sia δ il minimo comune multiplo dei δ_i e dei ϵ_i. Sia $F_{-\infty}(x)$ il risultato che si ottiene sostituendo in $F(x)$ tutte le occrrenze di atomi nella forma A e B con $true$ e $false$ rispettivamente. Analogamente si costruisce $F_{\infty}(x)$, dove però gli atomi nella forma A vengono sostituiti con $false$ e quelli nella forma B con $true$. Se il numero degli atomi di tipo A supera il numero degli atomi di tipo B si sostituisca $\exists\, x\, F(x)$ con

$$F^{-\infty} = \bigvee_{j=1}^{\delta} F_{-\infty}(j) \vee \bigvee_{j=1}^{\delta} \bigvee_{b_i} F(b_i + j)$$

Altrimenti si sostuisca con

$$F^{\infty} = \bigvee_{j=1}^{\delta} F_{\infty}(-j) \vee \bigvee_{j=1}^{\delta} \bigvee_{a_i} F(a_i - j)$$

A questo punto non resta che effettuare una semplificazione raccogliendo termini simili.

2.1 Complessità computazionale

Si mostri ora che, in un senso che verrà chiarito successivamente, se n è la dimensione della formula in ingresso, allora la formula equivalente senza variabili non potrà avere dimensione maggiore di $2^{2^{2^{pn}}}$, per qualche costante $p > 1$. Il procedimento seguito si deve a Derek C. Oppen.[3]

Una ulteriore interessante osservazione non rigorosa è la seguente: Michael J. Fischer e Michael O. Rabin[4] hanno trovato un bound inferiore per la complessità di una versione non deterministica dell'algoritmo, e tale bound risulta avere un esponenziale in meno. Dunque, siccome algoritmi deterministici che emulano algoritmi non deterministici introducono generalmente un esponenziale nella complessità, risulta auspicabile che il bound superiore trovato non sia migliorabile.

Sarà messa in relazione la crescita del numero degli atomi e la grandezza delle costanti con il numero dei coefficienti distinti che appaiono. Si cominci mostrando il seguente risultato preliminare.

Lemma 1. *Si consideri la formula*

$$Q_m x_m Q_{m-1} x_{m-1} \dots Q_2 x_2 Q_1 x_1\, F(x_1, x_2, \dots, x_m)$$

dove $Q_i = \exists$ oppure $Q_i = \forall$ e F è una formula senza quantificatori. Sia c_k la somma del numero di interi positivi distinti che appaiono negli atomi della forma $\delta_i \mid t$ e $\epsilon_i \nmid t$ e del numero dei coefficienti distinti delle variabili nella formula

$$F_k = Q_m x_m Q_{m-1} x_{m-1} \dots Q_{k+1} x_{k+1}\, F'_k(x_{k+1}, \dots, x_m)$$

prodotta dopo la k-esima iterazione. Analogamente sia s_k il massimo dei valori assoluti delle costanti intere, compresi i coefficienti della variabili. Infine sia a_k il numero totale degli atomi in F_k.

Allora valgono le seguenti relazioni:

$$c_1 \leq c^4 \qquad s_1 \leq s^{4c} \qquad a_1 \leq a^4 s^{2c}$$

[3] Derek C. Oppen. "A superexponential upper bound on the complexity of Presburger arithmetic". In: *J. Comput. System Sci.* 16.3 (1978), pp. 323–332. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90021-1.

[4] Michael J. Fischer e Michael O. Rabin. "Super-Exponential Complexity of Presburger Arithmetic". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition.* A cura di Bob F. Caviness e Jeremy R. Johnson. Vienna: Springer Vienna, 1998, pp. 122–135. ISBN: 978-3-7091-9459-1.

Dimostrazione. Siano a', a'', a''' il numero degli atomi dopo gli step 1 e 2 e 3, assumendo che a sia il numero degli atomi prima dell'esecuzione dell'algoritmo. Analogamnete si definiscano c', c'', c''' e s', s'', s'''. Si ripercorrano ora i passi dell'algoritmo, considerando mano a mano delle stime per tali valori.

Step 1. L'eliminazione delle negazioni logiche non altera nè c' nè s' nè a', l'elminazione dei simboli relazionali che non sono $|$, $\nmid$ o $<$ potrebbe raddoppiare il numero di atomi e potrebbe incrementare di 1 il massimo dei valori assoluti delle costanti che non appaiono come coefficienti dell variabili. Il numero di atomi con simboli relazionali $|$ o $\nmid$ resta al massimo a, dunque, una volta terminato il primo step dell'algoritmo si è nella seguente situazione:

$$a' \leq 2a \qquad s' \leq s+1 \qquad c' \leq c$$

Step 2. Sostituire x con $\delta' x$ potrebbe modificare il valore di s', il caso peggiore si verifica quando un atomo contiene sia il termine x (con coefficiente 1) che il termine s'. Il termine costante s' diventa $\delta' s'$, dove δ' è il minimo comune multiplo dei coefficienti della x. Siccome ci sono al massimo c coefficienti distinti della x, e ognuno di essi vale al massimo s, allora $\delta' \leq s^c$. Dunque $s'' \leq s^c s' \leq (s+1)^{c+1}$.

Anche il valore di c'' può venire alterato, ci sono al massimo $c-1$ variabili oltre alla x con coefficienti diversi da ogni coefficiente della x, inoltre ci sono al massimo c coefficienti c coefficienti distinti per la x. Dunque, c' può crescere al massimo fino a $c(c-1)+2$, dove $+2$ è dovuto da un $+1$ per l'eventuale nuovo coefficiente della x (che diventa 1) e un $+1$ è dovuto dalla costante δ' in $\delta' \mid x$. Infine questo step incrementa di 1 il numero di atomi, riassumendo si ha dunque il seguente bilancio.

$$a'' \leq 2a+1 \qquad s'' \leq (s+1)^{c+1} \qquad c'' \leq c^2$$

Step 3. Si consideri prima a''', il numero degli atomi in $\bigvee_{j=1}^{\delta} F_{-\infty}(j)$ è al massimo $\delta(a+1)$ siccome tutti gli atomi con il simbolo relazionale $<$ sono sostituiti da $true$ o $false$ e siccome ci sono al massimo $a+1$ atomi della forma $\delta_i \mid x + d_i$ o $\epsilon_i \nmid x + e_i$. A questo punto, grazie agli step 1 e 2, il numero di termini b_i è al massimo a, inoltre ci sono al massimo $2a+1$ atomi in $F(b_i+j)$. Quindi il numero di atomi in $\bigvee_{j=1}^{\delta} \bigvee_{b_i} F(b_i+j)$ è dominato superiormente da $\delta a(2a+1)$ e il numero di atomi a''' in $F^{-\infty}$ è al massimo $\delta(2a^2+2a+1) \leq \delta a^4$, per $a > 1$.

Occorre trovare ora bound superiore per δ, ogni costante δ_i o ϵ_i che appare in atomi della forma $\delta_i \mid x + d_i$ o $\epsilon_i \nmid x + e_i$ è il prodotto di due interi α e β, dove $\alpha \leq s$ e $\beta \mid \delta'$, ciò segue dal passo 2. Ci sono al massimo c valori di α distinti, quindi il minimo comune multiplo δ di tutti i δ_i e ϵ_i è al massimo $s^c \delta'$. Dunque $\delta \leq s^{2c}$ e $a''' \leq a^4 s^{2c}$.

La semplificazione dovuta al raccoglimento dei termini simili potrebbe alterare sia s''' che c''', la costante più grande potrebbe diventare $2s'' + 2^{2c} \leq 2(s+1)^{c+1} + s^{2c} \leq 3(s+1)^{2c}$. Un argomento simile a quello dato al passo 2 di questa dimostrazione fornisce una stima superiore per c''', ovvero $c''' \leq c^4$. Riassumendo:

$$a''' \leq a^4 s^{2c} \qquad s''' \leq 3(s+1)^{2c} \qquad c''' \leq c^4$$

Tuttavia per i nostri scopi saranno sufficienti le seguenti:

$$a_1 \leq a^4 s^{2c} \qquad s_1 \leq s^{4c} \qquad c_1 \leq c^4$$

per $s, c > 2$

$\square$

Lemma 2. *Se $s, c > 2$, allora*

$$c_k \leq c^{4^k} \qquad s_k \leq s^{(4c)^{4^k}} \qquad a_k \leq a^{4^k} s^{(4c)^{4^k}}$$

Dimostrazione. Per induzione sul lemma precedente. □

Si supponga dunque ora che sia data una frase di lunghezza n la quale codifica

$$Q_m x_m Q_{m-1} x_{m-1} \dots Q_2 x_2 Q_1 x_1 \, F(x_1, x_2, \dots, x_m)$$

Si desidera trovare un bound superiore allo spazio richiesto dalla formula senza quantificatori F_m. Si può assumere che $m \leq n, c \leq n, a \leq n, s \leq n$, per ogni k lo spazio richiesto per immagazzinare F_k è stimato dall'alto dal prodotto del numero degli atomi a_k in F_k, il massimo numero $m+1$ di costanti per atomo, la massima quantità di spazio s_k richiesta per immagazzinare ogni costante e una qualche costante q. Si osservi che il fattore q è dovuto ai vari operatori logici e aritmetici. Dunque lo spazio per immagazzinare F_k è stimato superiormente da

$$q \cdot n^{4^n} \cdot n^{(4n)^{4^n}} \cdot (n+1) \cdot n^{(4n)^{4^n}} \leq 2^{2^{2^{pn}}}$$

per una qualche costante $p > 1$. Si afferma inoltre che il bound superiore della complessità temporale dell'algoritmo è dominato dal quadrato del tempo richiesto per generare la F_k più lunga. Dunque il bound spaziale appena ottenuto è in realtà anche un bound temporale.

2.2 Dall'algoritmo all'implementazione

La versione dell'algoritmo che verrà implementata prevede alcune assunzioni col solo scopo di semplificare l'implementazione. Innanzitutto si considera già effettuato un primo passaggio di semplificazione, ovvero che la formula in ingresso φ sia in forma normale negativa e che siano stati rimossi i simboli non facenti strettamente parte dell'aritmetica di Presburger. Assumiamo cioè che φ sia congiunzione e disgiunzione dei seguenti tipi di letterali:

$$0 = t \qquad \neg(0 = t) \qquad 0 < t \qquad D_k(t) \qquad \neg D_k(t)$$

Diremo che φ in tale forma è una **formula ristretta**.

2.2.1 Normalizzazione dei coefficienti

Sia quindi $\exists\, x \,.\, \varphi$ con φ formula ristretta, il primo passaggio consiste nel trasformare φ in una formula dove il coefficiente della x sia sempre lo stesso. Per fare questo è sufficiente calcolare il minimo comune multiplo l di tutti i coefficienti di x ed effettuare i seguenti passi:

- Per le equazioni e le equazioni negate, rispettivamente nella forma $0 = t$ e $\neg(0 = t)$, si moltiplica t per l/c, dove c indica il coefficiente della x.
- Analogamente, per i predicati di divisibilità $k \mid t$ e i predicati di divisibiltà negati $k \nmid t$ si moltiplica sia t che k per l/c, sempre dove c indica il coefficiente della x.
- Per le diseguaglianze $0 < t$ si moltiplica t per il valore assoluto l/c, dove ancora un volta c indica il coefficiente della x.

Quindi ora tutti i coefficienti della x in φ sono $\pm l$, passiamo ora a considerare la seguente formula equivalente:

$$\exists\, x. \, (D_l(x) \land \psi)$$

dove ψ è ottenuta da φ sostituendo $l \cdot x$ con x. Dunque la formula $\varphi' = l \mid x \land \psi$ è una formula ristretta dove i coefficienti della x sono ± 1.

2.2.2 Costruzione di $\varphi'_{-\infty}$

Si definisce la nuova formula $\varphi'_{-\infty}$ ottenuta partendo da φ' e sostituendo tutte le formule atomiche α con $\alpha_{-\infty}$ secondo la seguente tabella:

α	$\alpha_{-\infty}$
$0 = t$	falso
$0 < t$ con $1 \cdot x$ in t	falso
$0 < t$ con $-1 \cdot x$ in t	vero
ogni altra formula atomica α	α

2.2.3 Calcolo dei boundary points

Ad ogni letterale $L[x]$ di φ' contenente la x che non è un predicato di divisibilità associamo un intero, detto **boundary point**, nel seguente modo:

Tipo di letterale	Boundary point
$0 = x + t$	il valore di $-(t + 1)$
$\neg(0 < x + t)$	il valore di $-t$
$0 < x + t$	il valore di $-t$
$0 < -x + t$	niente

Si osserva come nel caso la formula φ contenga più variabili da eliminare allora i valori nella colonna di destra possano dipendere da altre variabili. Chiamiamo B-set l'insieme di questi boundary points.

2.2.4 Eliminazione del quantificatore

Quest'ultimo passaggio è semplicemente l'applicazione della seguente equivalenza:[5]

$$\exists\, x \,.\, \varphi'[x] \longleftrightarrow \bigvee_{j=1}^{m} \Big(\varphi'_{-\infty}[j] \vee \bigvee_{b \in B} (\varphi'[b + j]) \Big)$$

dove φ' è la formula ristretta in cui i coefficienti della x sono sempre ± 1, m è il minimo comune multiplo di tutti i k dei predicati di divisibilità $d \mid t$ che appaiono in φ' tali che appaia la x in t e infine B è il B-set relativo a φ'. Considerando quindi il lato destro della precedente equivalenza si ha una formula priva del quantificatore esistenziale e si ha dunque ottenuto ciò che si voleva.

[5] D. C. Cooper. "theorem proving in arithmetic without multiplication". In: *machine intelligence 7* (1972), pp. 91–99. URL: `http://citeseerx.ist.psu.edu/showciting?cid=697241`.

3 Implementazione

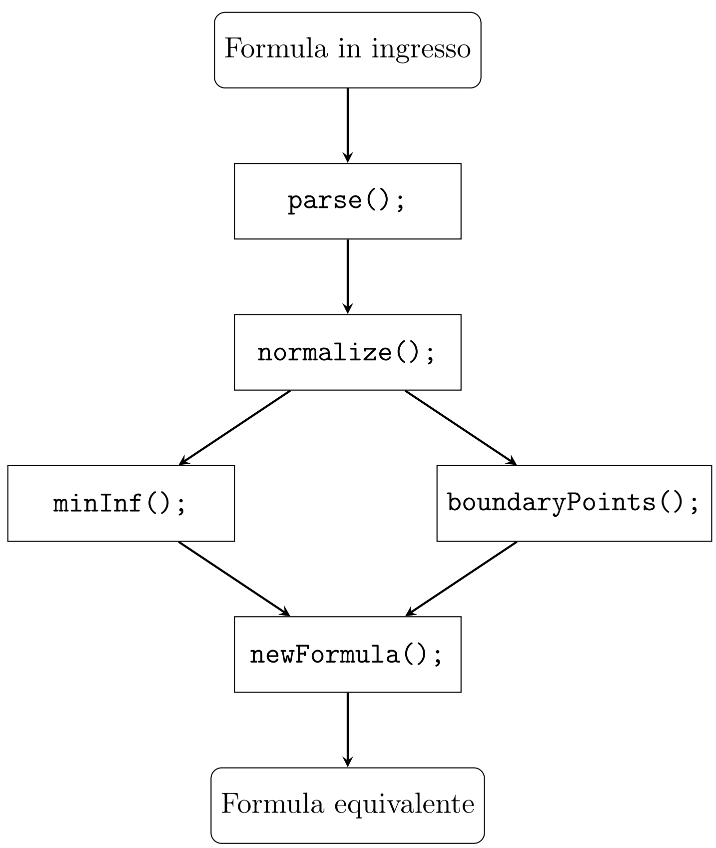
Il software è stato scritto nel linguaggio C rispettando lo standard C99,[6] in questo capitolo verrà effettuata una discussione riguardo l'implementazione. Occorre precisare subito che, a causa dell'obiettivo di questo software, ovvero interfacciarsi ad MCMT, è stata possibile una ulteriore semplificazione nelle assunzioni della forma dell'input. In sostanza la formula in ingresso deve avere la forma

$$\varphi(x) = \exists\, x.\ \bigwedge_{i=1}^{n} a_i(x)$$

dove $a_i(x)$ sono formule atomiche, ovvero della forma $t_i < t_i'$, $k \mid t_i$ o $t_i' = t_i'$, dove t_i e t_i' sono termini e $k \in \mathbb{N} \setminus \{0, 1\}$.

3.1 Struttura del sorgente

L'algoritmo è stato suddiviso in svariate procedure, implementate come singole funzioni in C, è possibile eseguire l'intero algoritmo chiamando la funzione `char* cooperToStr(char* wff, char* var)`, dove `wff` è una formula ben formata (well-formed formula) nel linguaggio SMT-LIB[7] e `var` è la variabile da eliminare. Naturalmente la funzione restituisce la formula equivalente priva della variabile. Si rimanda a più tardi la discussione della forma esatta che deve avere la formula in ingresso.



La funzione `cooperToStr` effettua quindi a sua volta delle chiamate a varie funzioni, si è cercato per quanto possibile di mantenere la suddivisione di queste sotto-procedure fedele alla descrizione dell'algoritmo svolta precedentemente.

[6] ISO. *ISO C Standard 1999*. Rapp. tecn. ISO/IEC 9899:1999 draft. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[7] Clark Barrett and Pascal Fontaine and Cesare Tinelli. *SMT-LIB*. ver. 2.6. 18 Giu. 2017. URL: `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf`.

Prima di spiegare il comportamento delle singole funzioni occorre accennare che l'oggetto principale manipolato dal programma è l'albero sintattico stesso della formula. Per ottenere ciò si è creato un tipo strutturato chiamato `t_syntaxTree` ad hoc. Si rimanda a più tardi una discussione dettagliata del tipo in questione.

La funzione che ha quindi il compito di effettuare il parsing è `t_syntaxTree* parse(char* wff)`, ed è questo appena introdotto il tipo che ritorna.

Il passo successivo al parsing è la normalizzazione della formula, cioè la generazione della formula $\varphi' = D_l(x) \wedge \psi$, dove i coefficienti della variabile da eliminare sono diventati 1. La segnatura di tale funzione è `void normalize(t_syntaxTree* tree, char* var)`.

Le funzioni `t_syntaxTree* minInf(t_syntaxTree* tree, char* var)` e `t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var)`, come è facile evincere, generano rispettivamente $\varphi'_{-\infty}$ e l'insieme dei boundary points.

Infine `t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var)` genera la formula equivalente a partire da $\varphi'_{-\infty}$ e dalla formula normalizzata. È al suo interno che viene effettuata la chiamata a `boundaryPoints`.

Esiste inoltre un ulteriore passo opzionale non facente parte dell'algoritmo di Cooper, la funzione `void simplify(t_syntaxTree* t)`, che può essere chiamata passando come argomento l'output di `newFormula()`, effettua una rozza semplificazione della formula. Verrà discusso successivamente in dettaglio cosa si intende.

3.2 Analisi delle procedure

Quella che viene presentata qui è un'analisi dettagliata del codice sorgente del programma riga per riga, si è deciso di seguire il più possibile il flusso di esecuzione del programma, in modo da evidenziare i passi dell'algoritmo.

3.2.1 Funzione `cooperToStr`

```c
char* cooperToStr(char* wff, char* var) { //Elimina var dalla formula
  t_syntaxTree* tree, *minf, *f;
  char* str;

  tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
  normalize(tree, var); //Trasforma l'albero di tree
  //printf("\nNormalizzato %s\n\n", treeToStr(tree));
  minf = minInf(tree, var); //Restituisce l'albero di φ−∞
  //printf("\nMininf %s\n\n", treeToStr(minf));
  //printf("\nbPts %s\n\n", treeToStr(boundaryPoints(tree, var)));
  f = newFormula(tree, minf, var); //Restituisce la formula equivalente
  //printf("\nFormula equivalente %s\n\n", treeToStr(f));
  simplify(f); //opzionale
  adjustForYices(f);
  str = treeToStr(f); //Genera la stringa a partire dall'albero

  recFree(tree); //Libera la memoria
  recFree(minf);
  recFree(f);

  return str;
}
```

Alla luce di quanto detto precedentemente il funzionamento di `cooper` risulta autoesplicativo. É quindi arrivato il momento di esporre la segnatura completa del tipo composto `t_syntaxTree`.

3.2.2 Segnatura di `t_syntaxTree`

```c
typedef struct t_syntaxTree { //Per rappresentare gli alberi sintattici
  char nodeName[16]; //Lunghezza massima scelta arbitrariamente
  int nodesLen; //Numero di figli dell'albero
  struct t_syntaxTree** nodes; //Puntatore
} t_syntaxTree;
```

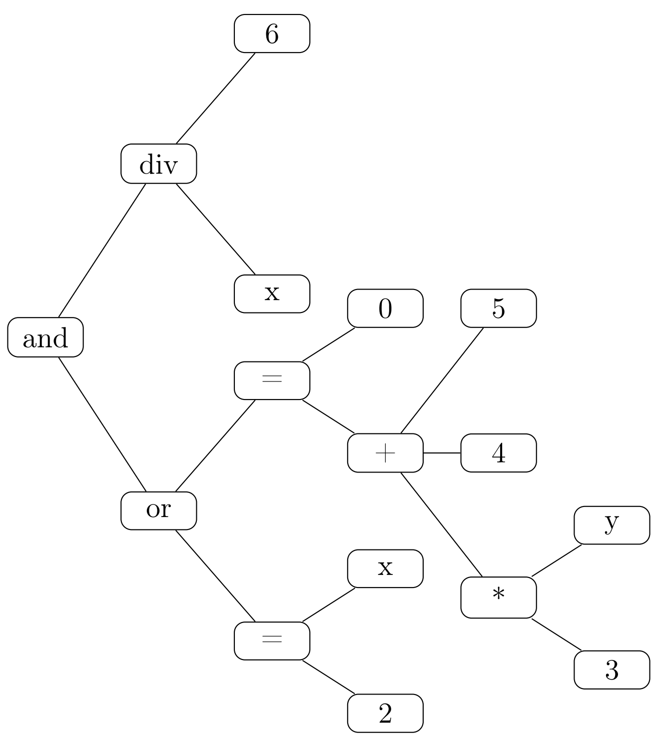
Trattasi di un record definito ricorsivamente avente 3 campi:

- `char nodeName[16]` è una stringa di lunghezza fissata posta arbritrariamente a 16 caratteri, è il nome del nodo nell'albero sintattico.

- `int nodesLen` è il numero di figli del nodo in questione

- `t_syntaxTree** nodes` è un array di puntatori ad altri nodi

Si consideri la formula in pseudolinguaggio $((2 = x) \wedge (3y+4+5 = 0)) \vee (x \equiv_6 0)$, in linguaggio SMT-LIB essa corrisponde a `(and (or (= 2 x) (= (+ (* 3 y) 4 5) 0)) (div x 6))` e la sua rappresentazione tramite il tipo composto appena definito è chiarificata dal segente diagramma.



Le foglie dell'albero sono semplicemente nodi con l'attributo `nodesLen` valente 0, in tal caso è irrilevante il contenuto del campo `nodes`. Si approfitta di questo momento per sottolineare l'importanza di una opportuna funzione di deallocazione di questa struttura.

3.2.3 Funzione recFree

```c
void recFree(t_syntaxTree* tree) { //Dealloca ricorsivamente tutto l'albero
  for (int i=0; i<tree->nodesLen; i++) {
    recFree(tree->nodes[i]);
  }

  free(tree->nodes);
  free(tree);
}
```

La natura ricorsiva del tipo `t_syntaxTree` rende notevolmente semplice la scrittura di una funzione ricorsiva per la liberazione della memoria, come è semplice intuire tale funzione effettua una visita in profondità dell'albero deallocando nodo per nodo.

Si passi ora a considerare due funzioni speculari, la funzione `t_syntaxTree* parse(char* wff)` che trasforma una stringa nel corrispettivo albero sintattico e la funzione `char* treeToStr(t_syntaxTree* tree)` che realizza l'esatto opposto.

3.2.4 Funzione parse

```c
t_syntaxTree* parse(char* wff, int strict) { //Funzione principale di parsing
  char* wffSpaced = malloc(sizeof(char)); //Sarà la stringa come la sringa in ingresso ma
                                          //con spazi tra i token
  wffSpaced[0] = wff[0];
  int j = 1;

  for (int i = 1; i < strlen(wff) + 1; i++) { //Scorro sui caratteri della stringa in
                                              //ingresso
    if (wff[i - 1] == '(') {
      wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
      wffSpaced[j] = ' ';
      wffSpaced[j + 1] = wff[i];
      j += 2;
    }

    else if (wff[i + 1] == ')') {
      wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
      wffSpaced[j] = wff[i];
      wffSpaced[j + 1] = ' ';
      j += 2;
    }

    else {
      wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 1));
      wffSpaced[j] = wff[i];
      j++;
    }
  }

  char* token; //Stringa di supporto contenente un singolo token
  int nTokens = 1; //Numero dei token
```

```c
  char** tokens = malloc(sizeof(char *)); //Array dei tokens
  tokens[0] = strtok(wffSpaced, " "); //Inserisco il primo token

  while ((token = strtok(NULL, " ")) != NULL) { //Finchè ce ne sono ne aggiungo
    nTokens++;
    tokens = realloc(tokens, sizeof(char *) * nTokens);
    tokens[nTokens - 1] = token;
  }

  int countPar = 0; //Contatore delle partentesi aperte incontrate finora

  for(int i=0; i<nTokens; i++) { //Scorro sui tokens
    for(int j=0; j<strlen(tokens[i]); j++) //Scorro sul singolo token
      if(tokens[i][j] == ')' && j!= 0) //Se ")" non è il primo carattere di un token
        ERROR("Parsing error: every S-expression must \
have a root and at least an argument");
    if (tokens[i][0] == '(') countPar++; //Incremento il contatore
    if (tokens[i][0] == ')') countPar--; //Decremento il contatore
  }

  if (countPar != 0) //Alla fine ogni parentesi aperta deve essere stata chiusa
    ERROR("Parsing error: the number of parentheses is not even");

  t_syntaxTree* syntaxTree = buildTree(0, tokens);

  if (strict) checkTree(syntaxTree); //chiama exit() se l'albero non va bene,
                                     //viene effettuato solo se strict != 0

  free(wffSpaced);
  free(tokens);

  return syntaxTree;
}
```

La funzione `parse` si appoggia alla funzione `buildTree`, è in quest'ultima la funzione, ancora una volta ricorsiva, dove avviene la vera e propria costruzione dell'albero. Essa prende in ingresso i token che compongono la stringa in ingresso e restituisce l'albero, la parte di suddivisione in token viene effettuata (insieme ad altre questioni di gestione della memoria) da `parse`. Tali funzioni prevedono che la stringa in ingresso rispetti esattamente la sintassi stabilita, e che inoltre, a causa della scelta arbitraria di porre 16 caratteri come lunghezza del campo `nodeName` non siano presenti token più lunghi.

3.2.5 Funzione `treeToStr`

```c
char* treeToStr(t_syntaxTree* tree) { //Funzione inversa di parse(), in realtà è un wrapper
                                      //non ricorsivo di recTreeToStr()
  char* str=malloc(sizeof(char));
  str[0] = '\0'; //Le stringhe finiscono con '\0' in C
  recTreeToStr(tree, &str, 1);
  return str;
}
```

Si consideri ora la funzione speculare `treeToStr`, anch'essa si appoggia a sua volta ad un'altra funzione, ovvero `recTreeToStr`, è in quest'ultima che avviene la trasformazione da albero in stringa, rendendo quindi `treeToStr` funge solamente da una funzione helper.

```c
int recTreeToStr(t_syntaxTree* t, char** str, int len) { //Trasforma un albero sintattico in
                                                          //una stringa
  if (t->nodesLen == 0) { //Se il nodo attuale non ha figli
    int nLen = len + strlen(t->nodeName); //Aggiorna la lunghezza della stringa prodotta
    *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
    strcat(*str, t->nodeName); //Aggiungi alla stringa il nome del nodo
    return nLen;
  }

  else { //Se invece il nodo attuale ha figli
    int nLen = len + strlen(t->nodeName) + 1; //Aggiorna la lunghezza della stringa prodotta
                                              //considerando anche la parentesi "("
    *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
    strcat(*str, "("); //Aggiungi alla stringa "(" ...
    strcat(*str, t->nodeName); //... e il nome del nodo

    for (int i=0; i<t->nodesLen; i++) { //Per ogni figlio del nodo attuale
      nLen++; //A causa dello spazio " "
      *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
      strcat(*str, " "); //Aggiungi lo spazio
      nLen = recTreeToStr(t->nodes[i], str, nLen); //Scarica ricorsivamente
    }

    nLen++; //A causa della parentesi ")"
    *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
    strcat(*str, ")"); //Aggiungi la parentesi ")"

    return nLen; //Ritorna quanti caratteri è diventata adesso, utile per la ricorsione
  }
}
```

Si ritorni ora a considerare i passi principali dell'algoritmo, così come sono esposti nella funzione `cooper`, dopo quanto detto finora rimane da considerare l'implementazione effettiva dell'algoritmo.

```c
  tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
  normalize(tree, var); //Trasforma l'albero di tree
  minf = minInf(tree, var); //Restituisce l'albero di φ−∞
  f = newFormula(tree, minf, var); //Restituisce la formula equivalente
  simplify(f); //opzionale
```

Ovvero rimangono da discutere le funzioni `normalize`, `minInf` e `newFormula`. Si adempia subito all'incombenza data dalla funzione `simplify`, di cui si ricorda fare parte di un passo opzionale.

3.2.6 Funzione `simplify`

```c
void simplify(t_syntaxTree* t) { //Semplifica l'albero (non è la migliore semplicifcazione)
  if (t->nodesLen != 0) { //Semplifico solo se il nodo attuale ha figli
```

```c
    int simplified = 0; //E' 0 se non ho effettuato semplificazioni (valore inizializzato)

    if (strcmp(t->nodeName, "and") == 0) { //Se il nodo attuale è un "and"
      for(int i=0; i<t->nodesLen; i++) { //E se scorrendo tra i figli ...
        if (strcmp(t->nodes[i]->nodeName, "false") == 0) { //... trovo un "false"
          simplified = 1; //Mi segno che ho effettuato una semplificazione

          for (int j=0; j<t->nodesLen; j++) //Dealloco tutto ...
            recFree(t->nodes[j]);

          strcpy(t->nodeName, "false"); //... e il nodo "and" diventa un "false"
          t->nodesLen = 0;
          break;
        }
      }
    }

    if (strcmp(t->nodeName, "or") == 0) { //Se il nodo attuale è un "or"
      for(int i=0; i<t->nodesLen; i++) { //E se scorrendo tra i figli ...
        if (strcmp(t->nodes[i]->nodeName, "true") == 0) { //... trovo un "true"
          simplified = 1; //Mi segno che ho effettuato una semplificazione

          for (int j=0; j<t->nodesLen; j++) //Dealloco tutto ...
            recFree(t->nodes[j]);

          strcpy(t->nodeName, "true"); //... e il nodo "or" diventa un "true"
          t->nodesLen = 0;
          break;
        }
      }
    }

    if (!simplified) //Se non ho effettuato semplificazioni ...
      for(int i=0; i<t->nodesLen; i++)
        simplify(t->nodes[i]); //... tento di semplificare i figli del nodo attuale
  }
}
```

Tale funzione effettua una visita in ampiezza dell'albero alla ricerca di nodi `or` o `and` ed effettuando una sostituzione di questi ultimi, rispettivamente con `true` e `false` nel caso almeno uno degli operandi di `or` sia `true` o uno degli operandi di `and` sia `false`. La visita in ampiezza viene troncata nel caso si verifichi uno di questi casi, in quanto il valore dell'espressione è già determinabile, risulta chiaro da questo il perchè della visita in ampiezza e non in profondità. Si faccia notare come questa funzione di semplificazione possa essere notevolmente migliorata aggiungendo la valutazione delle espressioni, tuttavia questa non banale aggiunta esula dallo scopo del progetto. In sostanza questa funzione fornisce un buon compromesso tra i benefici che porta il poter accorciare le espressioni generate dall'algoritmo e una ulteriore complessità aggiunta. Si noti infine come ancora una volta occorre prestare attenzione alla corretta deallocazione della memoria.

È giunto il momento di analizzare la funzione `normalize`, tale funzione si appoggia a sua volta alle funzione `getLCM` che a sua volta richiama `gcd` e `lcm`.

3.2.7 Funzioni `gcd` e `lcm`

```c
long int gcd(long int a, long int b) { //Massimo comun divisore
  return b == 0 ? a : gcd(b, a % b);
}

long int lcm(long int a, long int b) { //Minimo comune multiplo
  return abs((a / gcd(a, b)) * b);
}
```

Come è facile immaginare tali funzioni effettuano semplicemente il calcolo del massimo comun divisore e del minimo comune multiplo. Il primo viene svolto efficacemente dall'algoritmo di Euclide[8] mentre il secondo è dato banalmente dalla seguente.

$$lcm(a,b) = \frac{ab}{GCD(a,b)}$$

La funzione `getLCM` prende in ingresso l'albero sintattico e una variabile e restituisce il minimo comune multiplo di tutti i coefficienti di tale variabile presenti nella formula.

3.2.8 Funzione `getLCM`

```c
int getLCM(t_syntaxTree* tree, char* var) { //Restituisce il minimo comune multiplo dei
                                            //coefficienti di var che appaiaono in tutto
                                            //l'albero
  if (tree->nodeName[0] == '*') { //Se sono in un nodo "*" e ...
    if (strcmp(((t_syntaxTree *)tree->nodes[1])->nodeName, var) == 0) {
      //... se il secondo figlio è var (non può mai essere il primo!)
      return atoi(((t_syntaxTree *) tree->nodes[0])->nodeName); //ritorno il coefficiente
    }
  }

  int l = 1; //Inizializzo a 1 nel caso il nodo attuale sia senza figli

  for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli e ...
    l = lcm(l, getLCM((t_syntaxTree *) tree->nodes[i], var)); //... scarico ricorsivamente
  }

  return l;
}
```

`getLCM` visita ogni nodo dell'albero alla ricerca dei coefficienti della variabile `var`, ovvero cerca nodi della forma `(* c var)` dove appunto `var` è la variabile da eliminare mentre `c` è il coefficiente. È importante sottolineare come i nodi debbano avere il coefficiente in `.nodes[0]` e la variabile in `.nodes[1]`, cioè nodi della forma `(* var c)` non vengono correttatamente gestiti. Tale compromesso porta sicuramente ad una perdita di generalità che in questo caso particolare potrebbe anche essere evitata, ma lo stesso non si potrà dire in seguito, pertanto verrà assunto un tale input.

Risulta quindi ora utile discutere quale sia la forma esatta dell'input gestito dal programma, molte assunzioni che non portano a perdita di generalità sono state fatte, la maggior parte delle quali non evitabili a meno di dover scrivere molte funzioni ausiliarie di semplificazione. Si è scelta tale strada principalmente per due motivi:

[8] Euclid. *Euclid's Elements*. All thirteen books complete in one volume, The Thomas L. Heath translation, Edited by Dana Densmore. Green Lion Press, Santa Fe, NM, 2002, pp. xxx+499. ISBN: 1-888009-18-7; 1-888009-19-5.

- Già allo stato attuale il programma ha presentato molte difficoltà di natura tecnica non inerenti all'implementazione dell'algoritmo. Considerare una gamma più ampia di input avrebbe aggiunto una notevole complessità derivante dall'utilizzo del C senza nessuna libreria di supporto.
- L'obiettivo finale di questo progetto è quello di aggiungere una funzionalità al software MCMT,[9] scrivere una libreria di supporto per poter gestire più input avrebbe comportato la riscrittura di molto codice già presente in MCMT. Allo stesso tempo interfacciarsi al software preesistente avrebbe vincolato troppo il progetto, si è preferito un approccio intermedio in modo da poter comunque rendere questo software il più stand-alone possibile.

Si passi dunque ad esaminare la forma di albero più generale possibile in grado di essere manipolata dal programma; il nodo principale deve essere un `and` con almeno 1 figlio, tutti i figli di questo nodo devono essere obbligatoriamente `=`, `>` o `div`. Sia `=`, `>` che `div` devono avere esattamente 2 figli, il primo (cioè `.nodes[0]`) deve essere un polinomio lineare mentre il secondo (cioè `.nodes[1]`) deve essere una costante. Il polinomio lineare deve sempre essere della forma `(+ (* c1 x1) (* c2 x2) ... (* c3 x3))`, dove come prima, il primo figlio di `*` è una costante e il secondo è una variabile. La sintassi è questa anche nel caso una delle costanti sia uguale a 1.

Non è difficile convincersi che ogni albero può essere trasformato, con mere manipolazioni simboliche, in un albero di questa forma. Per rendere più chiaro quanto detto si consideri ad esempio la seguente formula:

$$\exists\, x\ .\ (2x + y = 3) \wedge (z < y) \wedge (x \equiv_2 0)$$

Tale formula trasformata in albero risulta equivalente alla seguente, si osservi come sono stati esplicitati anche i coefficienti ± 1 e come non siano presenti costanti tra i figli del nodo `+`.

```
(and (= (+ (* 2 x) (* 3 y)) 3)
     (> (+ (* 1 y) (* -1 z)) 0)
     (div (+ (* 1 x)) 2))
```

Ed ecco il listato relativo alla funzione `normalize` nella sua interezza, si osservi come esso prenda in ingresso l'albero sintattico della formula e la variabile da eliminare ma ritorni effettivamente `void`, ovvero si osservi come modifichi l'albero senza costruirne uno nuovo. Si faccia anche caso a come tale funzione sia fortemente vincolata alla rigida struttura sintattica che è stata supposta. Tale funzione oltre a normalizzare la formula (tutti i coefficienti della variabile da eliminare diventano 1) aggiunge anche un opportuno predicato di divisibilità come specificato nell'algoritmo.

3.2.9 Funzione `normalize`

```c
void normalize(t_syntaxTree* tree, char* var) { //Modifica i coefficienti della variabili
                                                //trasformandoli in 1 o -1
  int lcm = getLCM(tree, var);
  int c = lcm;

  for (int i=0; i<tree->nodesLen; i++) { //Scorro tra i figli della radice
    if (strcmp("=", tree->nodes[i]->nodeName) == 0 || //Se sono in un "=" o ...
        strcmp("div", tree->nodes[i]->nodeName) == 0) { //... in un "div"
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro tra gli addendi
```

[9] ghilardi, *mcmt: model checker modulo theories*, cit.

```
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
          c = atoi(addends[j]->nodes[0]->nodeName); //c è il coefficiente della variabile
      }

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro fra gli addendi
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) { //Se sono sulla variabile
          strcpy(addends[j]->nodeName, var); //In questo modo il coefficiente è 1
          free(addends[j]->nodes[0]);
          free(addends[j]->nodes[1]);
          addends[j]->nodesLen = 0;
        }
        else { //Se non sono sulla variabile
          sprintf(addends[j]->nodes[0]->nodeName,
                  "%d",
                  atoi(addends[j]->nodes[0]->nodeName)*lcm/c); //Ricalcolo i coefficienti
        }
      }

      //Ricalcolo anche i coefficienti del termine costante a secondo membro di "=" o "div"
      sprintf(tree->nodes[i]->nodes[1]->nodeName,
              "%d",
              atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/c);
    }

    else if (strcmp(">", tree->nodes[i]->nodeName) == 0) { //Se invcece sono su un ">"
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
          c = atoi(addends[j]->nodes[0]->nodeName); //Mi segno il coefficiente di var
        }
      }

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) { //Se sono sulla variabile
          if (c>0) strcpy(addends[j]->nodeName, ""); //Se il coefficiente è positivo non
                                                     //faccio niente
          else strcpy(addends[j]->nodeName, "-"); //Altrimenti lo cambio di segno
          strcat(addends[j]->nodeName, var);
          free(addends[j]->nodes[0]);
          free(addends[j]->nodes[1]);
          addends[j]->nodesLen = 0;
        }
        else { //Se non sono sulla variabile
          sprintf(addends[j]->nodes[0]->nodeName,
                  "%d",
                  atoi(addends[j]->nodes[0]->nodeName)*abs(lcm/c)); //Ricalcolo i
                                                                    //coefficienti
        }
      }
```

```c

      //Ricalcolo anche i coefficienti del secondo membro di ">"
      sprintf(tree->nodes[i]->nodes[1]->nodeName,
              "%d",
              atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/abs(c));
    }
  }

  tree->nodesLen++;
  tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
  tree->nodes[tree->nodesLen-1] = malloc(sizeof(t_syntaxTree));
  strcpy(tree->nodes[tree->nodesLen-1]->nodeName, "div");
  tree->nodes[tree->nodesLen-1]->nodesLen = 2;
  tree->nodes[tree->nodesLen-1]->nodes = malloc(sizeof(t_syntaxTree*) * 2);
  tree->nodes[tree->nodesLen-1]->nodes[0] = malloc(sizeof(t_syntaxTree));
  tree->nodes[tree->nodesLen-1]->nodes[1] = malloc(sizeof(t_syntaxTree));
  tree->nodes[tree->nodesLen-1]->nodes[0]->nodesLen = 0;
  tree->nodes[tree->nodesLen-1]->nodes[0]->nodes = NULL;
  tree->nodes[tree->nodesLen-1]->nodes[1]->nodesLen = 0;
  tree->nodes[tree->nodesLen-1]->nodes[1]->nodes = NULL;
  strcpy(tree->nodes[tree->nodesLen-1]->nodes[0]->nodeName, var);
  sprintf(tree->nodes[tree->nodesLen-1]->nodes[1]->nodeName, "%d", lcm);
}
```

La funzione `minInf`, come suggerisce il nome, riceve in ingresso la formula normalizzta φ' e restituisce $\varphi'_{-\infty}$. A differenza della funzione precedente essa restituisce effettivamente il nuovo albero.

3.2.10 Funzione `minInf`

```c
t_syntaxTree* minInf(t_syntaxTree* tree, char* var) { //Calcola φ-∞
  t_syntaxTree* nTree = recCopy(tree); //La funzione lavora su una copia dell'albero

  char minvar[16]; //La variabile con "-" davanti
  minvar[0] = '\0';
  strcpy(minvar, "-");
  strcat(minvar, var);

  for (int i=0; i<nTree->nodesLen; i++) { //Scorro sui figli della radice
    if (strcmp(">", nTree->nodes[i]->nodeName) == 0) { //Se sono in un ">"
      t_syntaxTree** addends = nTree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
        if (strcmp(addends[j]->nodeName, var) == 0) //Se l'addendo contiene la variabile
          strcpy(nTree->nodes[i]->nodeName, "false"); //L'addendo diventa "false"
        else if (strcmp(addends[j]->nodeName, minvar) == 0) //Se l'addendo contiene la
                                                            //variabile col segno "-"
          strcpy(nTree->nodes[i]->nodeName, "true"); //L'addendo diventa "true"
      }

      for (int j=0; j<nTree->nodes[i]->nodesLen; j++) //Scorro sui figli dei figli della
                                                      //radice
```

```
        recFree(nTree->nodes[i]->nodes[j]); //E dealloco tutto

      free(nTree->nodes[i]->nodes);
      nTree->nodes[i]->nodesLen = 0;
      nTree->nodes[i]->nodes = NULL;
    }

    else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) { //Se sono in un "="
      for (int j=0; j<nTree->nodes[i]->nodesLen; j++) //Scorro sui figli ...
        recFree(nTree->nodes[i]->nodes[j]); //... e dealloco tutto

      free(nTree->nodes[i]->nodes);
      nTree->nodes[i]->nodesLen = 0;
      nTree->nodes[i]->nodes = NULL;
      strcpy(nTree->nodes[i]->nodeName, "false"); //I nodi possono diventare solo "false"
    }
  }

  return nTree;
}
```

Prima di passare alla discussione della funzione `newFormula`, che effetivamente restituisce la formula equivalente senza variabile, è bene discutere di alcune altre funzioni a cui essa si appoggia, cioè `calcm` e `boundaryPoints`. La funzione `int calcm(t_syntaxTree* tree, char* var)` prende in ingresso l'albero della formula φ' e la variabile da eliminare e restituisce il minimo comune multiplo di tutti i coefficienti della x che appaiono nella formula, cioè calcola m dell'equivalenza di cui si è giò discusso.

$$\exists\, x \,.\, \varphi'[x] \longleftrightarrow \bigvee_{j=1}^{m} \left(\varphi'_{-\infty}[j] \vee \bigvee_{b \in B} (\varphi'[b+j]) \right)$$

3.2.11 Funzione `calcm`

```
int calcm(t_syntaxTree* tree, char* var) { //Calcolo il minimo comune multiplo di tutti i
                                           //coefficienti della variabile
  int m=1;

  for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli della radice
    if(strcmp(tree->nodes[i]->nodeName, "div") == 0) { //Se sono in un "div"

      if(strcmp(tree->nodes[i]->nodes[0]->nodeName, var) == 0) //Se trovo la variabile senza
                                                               //coefficiente (i.e. 1 o -1)
        m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName)); //Calcolo il m.c.m.

      else if(strcmp(tree->nodes[i]->nodes[0]->nodeName, "+") == 0) { //Altrimenti se trovo
                                                                       //un "+"
        for(int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Sorro tra gli addendi
          if (strcmp(tree->nodes[i]->nodes[0]->nodes[j]->nodeName, var) == 0) {
            m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
            break;
```

```
          }
        }
      }
    }
  }

  return m;
}
```

La funzione `t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var)` riceve ancora in ingresso l'albero sintattico della formula $\varphi'_{-\infty}$ e restituisce il B-set B della formula. Per semplicità di rappresentazione si è scelto di usare ancora come tipo per l'output sempre `t_syntaxTree`, dove però l'albero avrà come `.nodeName` la stringa arbitraria `"bPoints"`, tale scelta non ha nessun impatto e facilita semplicemente il debugging.

3.2.12 Funzione `boundaryPoints`

```
t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var) { //Calcolo dei bounday points
  char str[16];
  str[0] = '\0';
  t_syntaxTree* bPoints = malloc(sizeof(t_syntaxTree)); //Salvo i boundary points in un
                                                        //albero
  bPoints->nodes = NULL;
  strcpy(bPoints->nodeName, "bPoints"); //Nome utile solo per il debugging
  bPoints->nodesLen = 0;

  for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli della radice
    if (strcmp(tree->nodes[i]->nodeName, "=") == 0) { //Se sono in un "="
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
        if (strcmp(var, addends[j]->nodeName) == 0) { //Se trovo la variabile
          bPoints->nodesLen++; //Aggiungo un boundary point
          bPoints->nodes = realloc(bPoints->nodes,
                                   sizeof(t_syntaxTree *) * bPoints->nodesLen);
          t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
          bp->nodes = NULL;
          strcpy(bp->nodeName, "+");
          bp->nodesLen = 0;

          for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) { //Scorro sugli addendi
            if (strcmp(var, addends[k]->nodeName) != 0) { //Se non sono sulla variabile
              bp->nodesLen++; //Aggiungo comunque
              bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
              bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);

              sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
              strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
            }
          }

```

```
          bp->nodesLen++;
          bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
          bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
          bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
          bp->nodes[bp->nodesLen - 1]->nodes = NULL;
          sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
          strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);

          bPoints->nodes[bPoints->nodesLen-1] = bp;
          break;
        }
      }
    }

    if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(var, addends[j]->nodeName) == 0) {
          bPoints->nodesLen++;
          bPoints->nodes = realloc(bPoints->nodes,
                                   sizeof(t_syntaxTree *) * bPoints->nodesLen);
          t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
          bp->nodes = NULL;
          strcpy(bp->nodeName, "+");
          bp->nodesLen = 0;

          for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
            if (strcmp(var, addends[k]->nodeName) != 0) {
              bp->nodesLen++;
              bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
              bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
              sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
              strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
            }
          }

          bp->nodesLen++;
          bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
          bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
          bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
          bp->nodes[bp->nodesLen - 1]->nodes = NULL;
          sprintf(str, "%d", +atoi(tree->nodes[i]->nodes[1]->nodeName));
          strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);

          bPoints->nodes[bPoints->nodesLen-1] = bp;
          break;
        }
      }
    }
```

```
  }

  return bPoints;
}
```

Si discuta ora la funzione che restituisce la formula equivalente che poi `cooperToStr` ritorna, tale funzione è `t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var)`, essa non è altro che l'applicazione dell'equivalenza già esposta più volte. Prende in ingresso le forumule φ' e $\varphi'_{-\infty}$ e la variabile da eliminare, è al suo interno che vengono effettuate le chiamate a `boundaryPoints` e `calcm`.

3.2.13 Funzione `newFormula`

```
t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
  int m = calcm(minf, var); //Minimo comune multiplo dei coefficienti di var
  t_syntaxTree* val;
  char str[16];
  t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree)); //Nuovo albero che verrà restituito
  strcpy(nTree->nodeName, "or"); //La radice del nuovo albero è un "or"
  nTree->nodesLen = 0;
  nTree->nodes = NULL;

  t_syntaxTree* t;
  t_syntaxTree* bp;
  t_syntaxTree *bPts = boundaryPoints(tree, var); //Calcola i boundary points

  //La seguente è semplicemente una applicazione della formula del teorema
  for(int i=1; i<=m; i++) {
    nTree->nodesLen++;
    nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
    t = recCopy(minf);
    val = malloc(sizeof(t_syntaxTree));
    sprintf(str, "%d", i);
    strcpy(val->nodeName, str);
    val->nodesLen = 0;
    val->nodes = NULL;
    eval(t, var, val);
    recFree(val);
    nTree->nodes[nTree->nodesLen-1] = t;

    for(int j=0; j<bPts->nodesLen; j++) {
      nTree->nodesLen++;
      nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
      t = recCopy(tree);
      bp = recCopy(bPts->nodes[j]);
      sprintf(str, "%d", i+atoi(bp->nodes[bp->nodesLen-1]->nodeName));
      strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
      eval(t, var, bp);
      recFree(bp);

      nTree->nodes[nTree->nodesLen-1] = t;
```

```c
    }
  }

  recFree(bPts); //Non servono più, dealloco
  return nTree;
}
```

La funzione `newFormula` non fa altro che invocare `calcm` e `boundaryPoints` e generare l'albero della nuova formula equivalente, albero che poi ritorna. Eliminate le varie questioni di gestione della memoria quello che rimane è semplicemente un ciclo `for`. La funzione in realtà fa anche uso di un'ulteriore funzione di valutazione, ovvero una funzione che prende ingresso un albero, una variabile e un valore e va a sostituire il valore alla variabile.

Trattasi ovviamente della funzione `void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val)`, si osservi anche qui come tale funzione potrebbe essere resa più sofisticata aggiungendo una effettiva valutazione delle operazioni aritmetiche o logiche, ma come prima anche questo avrebbe aggiunto una ulteriore complessità al progetto, pertanto si è scelto di non proseguire in questa strada.

3.2.14 Funzione `eval`

```c
void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val) {
  for (int i=0; i<tree->nodesLen; i++) { //Scorro sui figli di tree
    if (strcmp(tree->nodes[i]->nodeName, var) == 0) { //Nel caso trovi var
      recFree(tree->nodes[i]); //Dealloco e ...
      tree->nodes[i] = recCopy(val); //... sostituisco con una copia di val
    }
    else { //Nel caso non trovi var potrei comunque ancora trovare var con "-" davanti
      char mvar[17] = "-"; //E' var con "-" davanti
      strcat(mvar, var);
      if (strcmp(tree->nodes[i]->nodeName, mvar) == 0) { //Se trovo var con "-" davanti
        recFree(tree->nodes[i]); //Dealloco tutto
        //Creo un nuovo nodo "-" contenente val come unico figlio
        tree->nodes[i] = malloc(sizeof(t_syntaxTree));
        strcpy(tree->nodes[i]->nodeName, "-");
        tree->nodes[i]->nodesLen = 1;
        tree->nodes[i]->nodes = malloc(sizeof(t_syntaxTree*));
        tree->nodes[i]->nodes[0] = recCopy(val);
      }
    }

    eval(tree->nodes[i], var, val); //Scarico ricorsivamente su tutti i figli
  }
}
```

4 Utilizzo

In questa sezione verranno forniti alcuni semplici esempi di utilizzo, innanzitutto si sottolinea come l'implementazione dell'algoritmo termini con la funzione `cooperToStr`, tutto quello che sta per essere esposto è al solo scopo di fornire una interfaccia che permetta di verificare il corretto funzionamento dell'algoritmo.

4.1 Programmi di esempio

Sono forniti assieme alla presente due programmi di esempio, il primo è `test.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "cooper.h"

int main(int argc, char** argv) {
  char* str;

  if (argc == 3) {
    str = cooperToStr(argv[1], argv[2]);
    printf("%s", str);
  }
  else
    printf("Numero errato di argomenti!\n");

  free(str);

  return 0;
}
```

Il secondo programma di esempio è `test2.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "cooper.h"

int main(int argc, char** argv) {
  char** array;
  int len;

  if (argc == 3) {
    array = cooperToArray(argv[1], argv[2], &len);

    for (int i=0; i<len; i++) {
      printf("%s\n", array[i]);
      free(array[i]);
    }

    free(array);
  }
  else
    printf("Numero errato di argomenti!\n");
```

```

  return 0;
}
```

Come si può notare tale programma si appoggia alla funzione `cooperToArray`, gemella di `cooper`.

```
char** cooperToArray(char* wff, char* var, int* len) {
  t_syntaxTree* tree, *minf, *f;
  char* buffer;
  char** array;

  tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
  normalize(tree, var); //Trasforma l'albero di tree
  minf = minInf(tree, var); //Restituisce l'albero di φ−∞
  f = newFormula(tree, minf, var); //Restituisce la formula equivalente
  simplify(f); //opzionale
  adjustForYices(f);

  *len = f->nodesLen;

  array = malloc(sizeof(char*) * *len);

  for (int i=0; i<*len; i++) {
    buffer = treeToStr(f->nodes[i]);
    array[i] = malloc(sizeof(char) * strlen(buffer));
    strcpy(array[i], buffer);
    free(buffer);
  }

  recFree(tree); //Libera la memoria
  recFree(minf);
  recFree(f);

  return array;
}
```

4.2 Il Makefile

```
SHELL := /bin/bash
PARAMS = -std=c99 -O3 -Wall -g #compila nello standard C99 e abilita tutti i warning
leak-check = yes #valgrind effettua una ricerca dei leak più accurata
track-origins = yes #valgrind fornisce più informazioni
wff = "(and (= (+ (* -2 x) (* 3 y)) 3) \
            (> (+ (* 5 x) (* 3 y)) 1) \
            (div (+ (* 2 x) (* 4 y)) 1))" #formula in ingresso
wff = "(and (div (+ (* 3 z)) 3) (= (+ (* 2 y) (* 3 x)) 2) (= (+ (* 2 x)) 4))"
wff= "(and (> (+ (* 1 x)) 5) (> (+ (* -1 x) (* 1 y)) 0) (div (+ (* 1 1) (* 2 x)) 3))"
#wff="(and (= (+ (* 2 a) (* 3 b) (* 4 c)) 3) (> (+ (* 3 x) (* 2 y)) 1) (= (+ (* 2 x) (* 4 y)) 3) (>
#wff="(and (= (+ (* 2 x) (* -1 y)) 3) (= (+ (* 4 a) (* 1 y)) 1) (> (+ (* -2 x) (* 1 y)) -10) (= (+
vars = "x y" #variabili presenti nella formula
var = "x" #variabile da eliminare
```

```

test: test.c cooper.o
	gcc $(PARAMS) test.c cooper.o -o test

test2: test2.c cooper.o
	gcc $(PARAMS) test2.c cooper.o -o test2

cooper.o: cooper.c cooper.h
	gcc $(PARAMS) -c cooper.c -o cooper.o

run: test #esegue test e restituisce il tempo impiegato
	@echo -e 'Elimino la variabile $(var) dalla seguente formula:\n$(wff) ---> \n'
	@time ./test $(wff) $(var)

run2: test2
	@time ./test2 $(wff) $(var)

eq: test eq.py #prende la formula e la variabile da eliminare, la elimina controlla con z3 se il r
	@python3 eq.py "$(formula)" "$(variables)" "$(guess)" #la prima in variables è quella da el

valgrind: test
	valgrind --track-origins=$(track-origins) \
		--leak-check=$(leak-check) ./test $(wff) $(var)

valgrind2: test2
	valgrind --track-origins=$(track-origins) \
		--leak-check=$(leak-check) ./test2 $(wff) $(var)

debug: test #esegue test col debugger gdb
	gdb --args test $(wff) $(var)

eval: test3 #valuta il valore della formula equivalente,
	   #funziona solo se ogni variabile è già stata eliminata
	./eval.scm "`./test3 $(wff) $(vars) | tail -n 1`"

clean:
	rm -f *.o
	rm -f test test2
	rm -f eq.smt
```

Si consideri ora il seguente `makefile`: É semplice immaginare cosa facciano le regole `run`, `run2`, `valgrind`, `debug` e `clean`. Ci si soffermi ora su `eval` e `eq`. La prima esegue semplicemente `test` con la formula in ingresso specificata nel `makefile` e cerca di valutare la formula equivalente generata tramite il seguente script in Guile Scheme.[10]

4.3 Valutazione ed equivalenza

```
#!/bin/guile \
-e main -s
```

[10] GNU. *GNU Ubiquitous Intelligent Language for Extensions (GUILE).*

```
!#

(use-modules (ice-9 format) (ice-9 eval-string))

(define (div a b)
  (if (= (remainder a b) 0) #t #f))

(define true #t)

(define false #f)

(define (main args)
  (let ((str (cadr args)))
    (format #t
            "\nInput: ~s\nEvaluated: ~s\n"
            str
            (if (eval-string str) "true" "false"))))
```

Tale script valuta semplicemente la formula equivalente, è stato scelto un linguaggio della famiglia Lisp in quanto condivide la sintassi con SMT-LIB e ciò rende la valutazione della formula una semplice chiamata alla funzione `eval-string`.

Si ricorda come ovviamente tale procedura non è un verifica della soddisfacibilità, cioè qualora fossero ancora presenti variabili nella formula equivalente allora tale script produrrebbe un errore. Per una verifica della soddsfacibilità si usi invece la regola `eq` del `makefile`. Tale regola esegue il seguente script Python.[11]

```python
#!/usr/bin/env python3
from sys import argv
from subprocess import run, PIPE


def main():
    if len(argv) != 4:
        print("Wrong arguments number!")
    else:
        formula = argv[1]
        variables = argv[2].split() #tutte le variabili
        var = variables[0] #la prima è quella da eliminare
        guess = argv[3]
        smt_source = ""

        for v in variables:
            if v is not var:
                smt_source += "(declare-const {} Int)\n".format(v)

        wff_out = run(["./test", formula, var], stdout=PIPE).stdout.decode()
        smt_source += "(assert (not (= {} {})))\n".format(wff_out, guess)
        smt_source += "(check-sat)\n"

        with open("eq.smt2", "w") as source:
```

[11] Python Software Foundation. *Python language*. Ver. 3.7.2. 2019. URL: `https://www.python.org/`.

```python
            print(smt_source, file=source)

        print("Verifico se {} e' equivalente a \n{}".format(guess, wff_out))
        print("Eseguo con z3 il file 'eq.smt2' contenente: \n{}".format(smt_source))

        result = run(["z3", "eq.smt2"], stdout=PIPE).stdout.decode()

        if "unsat" in result:
            print("Sono equivalenti")
        elif "error" in result:
            print("Errore di z3:\n{}".format(result))
        elif "sat" in result:
            print("Non sono equivalenti")


if __name__ == '__main__':
    main()
```

Tale script genera un opportuno sorgente per z3[12] e successivamente lo esegue (poi lo elimina), per esempio eseguendo

```
make eq formula="(and (> (+ (* 1 x)) 5) (> (+ (* -1 x) (* 1 y)) 0))"
        variables="x y" guess="(> y 6)"
```

viene generato, eseguito con z3 e infine eliminato il seguente programma:

```
(declare-const y Int)
(assert (not (= (or false (and (> (+ (+ 6)) 5) (> (+ (- (+ 6)) (* 1 y)) 0)
                    (= (mod (+ 6) 1) 0))) (> y 6))))
(check-sat)
```

A questo punto lo script Python valuta ciò che z3 restituisce e stampa se le formule sono equivalenti o meno.

4.4 Esempi

Vengono mostrati ora alcuni esempi di utilizzo della libreriatramite il comando `make eq`.

4.4.1 Esempio 1

Si voglia eliminare la variabile x dalla seguente formula:

$$x > 5 \wedge y < x$$

É immediato osservare che tale formula è equivalente a $y > 6$, pertanto eseguiamo il comando

```
make eq formula="(and (> (+ (* 1 x)) 5) (> (+ (* -1 x) (* 1 y)) 0))"
        variables="x y" guess="(> y 6)"
```

[12] Microsoft Research. *Z3*. Ver. 4.8.5. URL: `https://github.com/Z3Prover/z3/`.

Ovvero verifichiamo se quanto produce l'algoritmo di Cooper sia equivalente o meno a $y > 6$ (passato a `make eq` come parametro `guess`). Si ricorda che i nomi delle variabili contenute in `formula` vanno passate a `make eq` come parametro `variables` e che la prima di esse é quella che viene eliminata. Eseguendo questo comando otteniamo come voluto "Sono equivalenti". Si ribadisce come la forma di `formula` che si aspetta il programma debba rispettare quanto detto nei capitoli precedenti, ovvero:

- la radice della formula deve essere un `and`
- i figli della radice possono essere esclusivamente `>`, `=` o `div`
- ognuno di questi deve avere esattamente 2 figli: il primo deve essere un `+` e il secondo un numero (eventualmente con segno negativo)
- i figli di `+` possono essere solamente `*` e ognuno di questi deve avere esattamente due figli: il primo un numero (eventualmente con segno negativo) e il secondo una variabile (lunga massimo 16 caratteri).
- é possibile avere anche figli di `+` che siano `*` con entrambi i figli costanti, in tal caso peró il primo figlio deve essere necessariamente `1` o `-1`. Questo si rivela fondamentale per inserire costanti nelle formule con `div`

4.4.2 Esempio 2

É anche possibile verificare direttamente la veritá delle formule confrontando se esse sono equivalenti a `true` e `false`. Si voglia controllare se la seguente formula é vera:

$$x = 2k \wedge x \equiv_2 1$$

Tale formula, palesemente falsa, puó essere verificata con il comando

```
make eq formula="(and (= (+ (* 1 x) (* -2 k)) 0) (div (+ (* 1 1) (* 1 x)) 2))"
        variables="x k" guess="true"
```

Ossia si controlla se la formula é equivalente a `true`, che essendo falso, fa si che venga stampato che le formule non sono equivalenti.

4.4.3 Esempio 3

Analogamente al precedente esempio si verifica che invece la seguente formula é vera

$$x = 2k + 1 \wedge x \equiv_2 1$$

Il comando

```
make eq formula="(and (= (+ (* 1 x) (* -2 k)) 1) (div (+ (* 1 1) (* 1 x)) 2))"
        2variables="x k" guess="true"
```

restituisce "Non sono equivalenti". Si osservi come in questi ultimi due esempi si sarebbe potuto scambiare l'ordine delle variabili x e k in `variables`, ovvero eliminare k invece di x.

Indice

Riferimenti bibliografici

Clark Barrett and Pascal Fontaine and Cesare Tinelli. *SMT-LIB*. Ver. 2.6. 18 Giu. 2017. URL: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

Cooper, D. C. "theorem proving in arithmetic without multiplication". In: *machine intelligence 7* (1972), pp. 91–99. URL: http://citeseerx.ist.psu.edu/showciting?cid=697241.

Euclid. *Euclid's Elements*. All thirteen books complete in one volume, The Thomas L. Heath translation, Edited by Dana Densmore. Green Lion Press, Santa Fe, NM, 2002, pp. xxx+499. ISBN: 1-888009-18-7; 1-888009-19-5.

Fischer, Michael J. e Michael O. Rabin. "Super-Exponential Complexity of Presburger Arithmetic". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. A cura di Bob F. Caviness e Jeremy R. Johnson. Vienna: Springer Vienna, 1998, pp. 122–135. ISBN: 978-3-7091-9459-1.

ghilardi, silvio. *mcmt: model checker modulo theories*. http://users.mat.unimi.it/users/ghilardi/mcmt/. 2018.

GNU. *GNU Ubiquitous Intelligent Language for Extensions (GUILE)*.

ISO. *ISO C Standard 1999*. Rapp. tecn. ISO/IEC 9899:1999 draft. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

Microsoft Research. *Z3*. Ver. 4.8.5. URL: https://github.com/Z3Prover/z3/.

Oppen, Derek C. "A superexponential upper bound on the complexity of Presburger arithmetic". In: *J. Comput. System Sci.* 16.3 (1978), pp. 323–332. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90021-1.

Presburger, Mojżesz. "On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation". In: *Hist. Philos. Logic* 12.2 (1991). Translated from the German and with commentaries by Dale Jacquette, pp. 225–233. ISSN: 0144-5340. DOI: 10.1080/014453409108837187.

Python Software Foundation. *Python language*. Ver. 3.7.2. 2019. URL: https://www.python.org/.