

Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

23 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 quantificatori² 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.

2 L'algoritmo di Cooper

Si ha quindi che l'algoritmo ha in ingresso una formula del tipo $\exists x. \varphi$ e in uscita una formula equivalente senza il quantificatore esistenziale. Se si vogliono eliminare più quantificatori esistenziali basta reiterare l'algoritmo.

Si osserva come ovviamente ogni formula contenente quantificatori universali possa essere trasformata in una formula equivalente con soli quantificatori esistenziali. Pertanto non si ha una perdita di generalità ad assumere un input in tale forma.

2.1 Processo di semplificazione

In questo passaggio vengono effettuate le seguenti semplificazioni alla formula in ingresso φ :

- Tutti i connettivi logici composti, cioè che non sono \neg , \wedge o \vee , vengono sostituiti nella loro definizione in termini di \neg , \wedge o \vee .
- I predicati binari \geq e \leq vengono sostituiti con le loro definizioni (e.g. $s \leq t$ diventa $s < t + 1$).

¹silvio ghilardi. *mcmt: model checker modulo theories*. <http://users.mat.unimi.it/users/ghilardi/mcmt/>. 2018.

²Mojżesz 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: 10.1080/014453409108837187. URL: <https://doi-org.pros.lib.unimi.it:2050/10.1080/014453409108837187>.

- Le disequaglianze negate della forma $\neg(s < t)$ vengono sostituite con $t < s + 1$.
- Tutte le equazioni e le disequazioni vengono riscritte in modo da avere 0 nel lato sinistro ($s = t$ e $s < t$ diventano $0 = t - s$ e $0 < t - s$).
- Tutti gli argomenti dei predicati vengono sostituiti con la loro forma canonica.

Dopo aver applicato queste sostituzioni e aver trasformato la φ ottenuta in forma normale negativa possiamo dunque assumere che φ sia congiunzione e disgiunzione dei seguenti tipi di letterali:

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

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

2.2 Normalizzazione dei coefficienti

Assumiamo quindi che l'algoritmo riceva in ingresso $\exists x. \varphi$ con φ formula ristretta. Il primo passaggio consiste nel trasformare φ in una formula dove il coefficiente della x è 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 $D_k(t)$ e i predicati di divisibilità negati $\neg D_k(t)$ si moltiplica sia t che k per l/c , sempre dove c indica il coefficiente della x .
- Per le disequaglianze $0 < t$ 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:

$$\exists x. (D_l(x) \wedge \psi)$$

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

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

Definiamo una 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

2.4 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.

2.5 Eliminazione dei quantificatori

Quest'ultimo passaggio è semplicemente l'applicazione della seguente equivalenza:³

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

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 $D_k(t)$ 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.

³d. 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 Complessità computazionale

In questa sezione verrà formalizzata in modo rigoroso una versione equivalente dell'algoritmo di Cooper, la quale permetterà di ottenere una stima superiore della complessità. Si vedrà infatti 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 $2^{2^{pn}}$, per qualche costante $p > 1$. Questo fornisce un bound superiore alla complessità temporale.

Una ulteriore osservazione non rigorosa è la seguente; Fischer e Rabin⁴ 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 non possono che introdurre un esponenziale nella complessità, risulta auspicabile che il bound superiore trovato non sia migliorabile.

3.1 Formalizzazione dell'aritmetica di Presburger

Si definiscano i simboli dell'aritmetica di Presburger:

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

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 \mathcal{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 $(t_1 < t_2)$ o $(t_1 = t_2)$, dove t_1 e t_2 sono termini. Una formula è definita come segue:

- Un atomo 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 $(2, 3, \dots)$ e altri simboli non facente parti del linguaggio. Ciononostante essi potranno sempre essere sostituiti con una composizione dei simboli appena esposti, dunque non andranno ad inficiare la validità dell'argomento.

3.2 L'algoritmo di Cooper come procedura decisionale

L'algoritmo di Cooper, se iterato su di una frase in \mathcal{L} permette di eliminare tutti i quantificatori, e quindi di valutare la verità di tale frase. In tale senso può essere inteso come procedura decisionale. Vengono quindi mostrati i passaggi effettuati dall'algoritmo in una singola iterazione.

Conseriamo 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 $\neg \exists x \neg$.

⁴Michael 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.

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 $<$, $|$ 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 sostituisce $\exists x F(\delta'x)$ con $\exists x (F(x) \wedge \delta' | x)$. 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. $\delta_i | x + c_i$
- 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 occorrenze 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 sostituisca 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.

3.3 Analisi e stima della complessità

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 $\delta_i | t$ 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 delle variabili. Infine sia a_k il numero totale degli atomi in F_k .

Allora valgono le seguenti relazioni:

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

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. Analogamente si definiscano c', c'', c''' e s', s'', s''' . Si ripercorrono 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'eliminazione dei simboli relazionali che non sono $|$, \dagger o $<$ potrebbe raddoppiare il numero di atomi e potrebbe incrementare di 1 il massimo dei valori assoluti delle costanti che non appaiono come coefficienti delle variabili. Il numero di atomi con simboli relazionali $|$ o \dagger resta al massimo a , dunque, una volta terminato il primo step dell'algoritmo si è nella seguente situazione:

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

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 $\delta' | x$. Infine questo step incrementa di 1 il numero di atomi, riassumendo si ha dunque il seguente bilancio.

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

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 $\delta_i | x + d_i$ o $\epsilon_i \dagger x + e_i$. 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 $\delta_i | x + d_i$ o $\epsilon_i \dagger x + e_i$ è il prodotto di due interi α e β , dove $\alpha \leq s$ e $\beta | \delta'$, 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:

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

Tuttavia per i nostri scopi saranno sufficienti le seguenti:

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

per $s, c > 2$

□

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

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

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

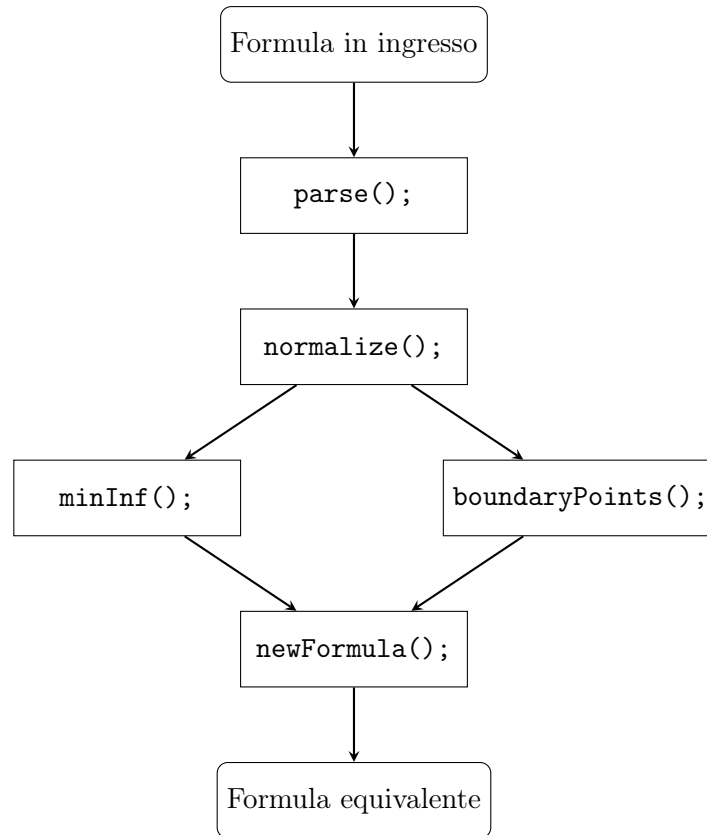
$$q \cdot n^{4^n} \cdot n^{(4n)^{4^n}} \cdot (n + 1) \cdot n^{(4n)^{4^n}} \leq 2^{2^{2^{pn}}}$$

per una qualche costante $p > 1$ Si affermi 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.

4 Implementazione

Il software è stato scritto nel linguaggio C rispettando lo standard C99,⁵ in questo capitolo verrà effettuata una discussione riguardo l'implementazione.

4.1 Struttura e design



L'algoritmo è stato suddiviso in svariate procedure, implementate come singole funzioni in C, è possibile eseguire l'intero algoritmo chiamando la funzione `char* cooper(char* wff, char* var)`, dove `wff` è una formula ben formata (well-formed formula) nel linguaggio SMT-LIB⁶ 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 `cooper` 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.

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.

⁵ISO. *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>.

⁶Clark 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>.

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.

4.2 Analisi del codice

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.

4.2.1 Funzione cooperToStr

```

653 char* cooperToStr(char* wff, char* var) { //Elimina var dalla formula
654     t_syntaxTree* tree, *minf, *f;
655     char* str;
656
657     tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
658     normalize(tree, var); //Trasforma l'albero di tree
659     //printf("\nNormalizzato %s\n\n", treeToStr(tree));
660     minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
661     //printf("\nMininf %s\n\n", treeToStr(minf));
662     //printf("\nbPts %s\n\n", treeToStr(boundaryPoints(tree, var)));
663     f = newFormula(tree, minf, var); //Restituisce la formula equivalente
664     //printf("\nFormula equivalente %s\n\n", treeToStr(f));
665     simplify(f); //opzionale
666     adjustForYices(f);
667     str = treeToStr(f); //Genera la stringa a partire dall'albero
668
669     recFree(tree); //Libera la memoria
670     recFree(minf);
671     recFree(f);
672
673     return str;
674 }
```

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

4.2.2 Segnatura di t_syntaxTree

```

8 typedef struct t_syntaxTree { //Per rappresentare gli alberi sintattici
9     char nodeName[16]; //Lunghezza massima scelta arbitrariamente
```

```

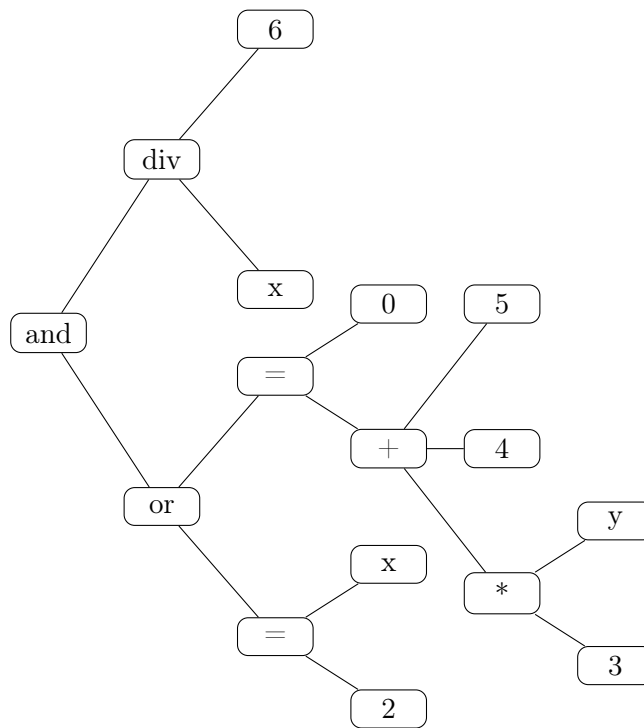
10  int nodesLen; //Numero di figli dell'albero
11  struct t_syntaxTree** nodes; //Puntatore
12  } t_syntaxTree;

```

Trattasi di un record definito ricorsivamente avente 3 campi:

- **char** nodeName[16] è una stringa di lunghezza fissata posta arbitrariamente 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 seguente 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.

4.2.3 Funzione recFree

```

285 void recFree(t_syntaxTree* tree) { //Dealloca ricorsivamente tutto l'albero
286     for (int i=0; i<tree->nodesLen; i++) {
287         recFree(tree->nodes[i]);
288     }
289
290     free(tree->nodes);
291     free(tree);
292 }

```

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.

4.2.4 Funzione parse

```

113 t_syntaxTree* parse(char* wff, int strict) { //Funzione principale di parsing
114     char* wffSpaced = malloc(sizeof(char)); //Sarà la stringa come la stringa in ingresso ma
115                                             //con spazi tra i token
116     wffSpaced[0] = wff[0];
117     int j = 1;
118
119     for (int i = 1; i < strlen(wff) + 1; i++) { //Scorro sui caratteri della stringa in
120                                             //ingresso
121         if (wff[i - 1] == '(') {
122             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
123             wffSpaced[j] = ' ';
124             wffSpaced[j + 1] = wff[i];
125             j += 2;
126         }
127
128         else if (wff[i + 1] == ')') {
129             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
130             wffSpaced[j] = wff[i];
131             wffSpaced[j + 1] = ' ';
132             j += 2;
133         }
134
135         else {
136             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 1));
137             wffSpaced[j] = wff[i];
138             j++;
139         }
140     }
141
142     char* token; //Stringa di supporto contenente un singolo token
143     int nTokens = 1; //Numero dei token
144     char** tokens = malloc(sizeof(char *)); //Array dei tokens
145     tokens[0] = strtok(wffSpaced, " "); //Inserisco il primo token
146
147     while ((token = strtok(NULL, " ")) != NULL) { //Finchè ce ne sono ne aggiungo
148         nTokens++;
149         tokens = realloc(tokens, sizeof(char *) * nTokens);
150         tokens[nTokens - 1] = token;
151     }
152
153     int countPar = 0; //Contatore delle parentesi aperte incontrate finora

```

```

154
155     for(int i=0; i<nTokens; i++) { //Scorro sui tokens
156         for(int j=0; j<strlen(tokens[i]); j++) //Scorro sul singolo token
157             if(tokens[i][j] == ')') && j!= 0) //Se ")" non è il primo carattere di un token
158                 ERROR("Parsing error: every S-expression must \
159 have a root and at least an argument");
160         if (tokens[i][0] == '(') countPar++; //Incremento il contatore
161         if (tokens[i][0] == ')') countPar--; //Decremento il contatore
162     }
163
164     if (countPar != 0) //Alla fine ogni parentesi aperta deve essere stata chiusa
165         ERROR("Parsing error: the number of parentheses is not even");
166
167     t_syntaxTree* syntaxTree = buildTree(0, tokens);
168
169     if (strict) checkTree(syntaxTree); //chiama exit() se l'albero non va bene,
170                                         //viene effettuato solo se strict != 0
171
172     free(wffSpaced);
173     free(tokens);
174
175     return syntaxTree;
176 }

```

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.

4.2.5 Funzione `treeToStr`

```

623 char* treeToStr(t_syntaxTree* tree) { //Funzione inversa di parse(), in realtà è un wrapper
624                                     //non ricorsivo di recTreeToStr()
625     char* str=malloc(sizeof(char));
626     str[0] = '\0'; //Le stringhe finiscono con '\0' in C
627     recTreeToStr(tree, &str, 1);
628     return str;
629 }

```

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.

```

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

```

```

597     return nLen;
598 }
599
600 else { //Se invece il nodo attuale ha figli
601     int nLen = len + strlen(t->nodeName) + 1; //Aggiorna la lunghezza della stringa prodotta
602                                           //considerando anche la parentesi "("
603     *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
604     strcat(*str, "("); //Aggiungi alla stringa "(" ...
605     strcat(*str, t->nodeName); //... e il nome del nodo
606
607     for (int i=0; i<t->nodesLen; i++) { //Per ogni figlio del nodo attuale
608         nLen++; //A causa dello spazio " "
609         *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
610         strcat(*str, " "); //Aggiungi lo spazio
611         nLen = recTreeToStr(t->nodes[i], str, nLen); //Scarica ricorsivamente
612     }
613
614     nLen++; //A causa della parentesi ")"
615     *str = realloc(*str, sizeof(char) * nLen); //Rialloca la stringa
616     strcat(*str, ")"); //Aggiungi la parentesi ")"
617
618     return nLen; //Ritorna quanti caratteri è diventata adesso, utile per la ricorsione
619 }
620 }

```

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.

```

682 tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
683 normalize(tree, var); //Trasforma l'albero di tree
684 minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
685 f = newFormula(tree, minf, var); //Restituisce la formula equivalente
686 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.

4.2.6 Funzione `simplify`

```

550 void simplify(t_syntaxTree* t) { //Semplifica l'albero (non è la migliore semplificazione)
551     if (t->nodesLen != 0) { //Semplifico solo se il nodo attuale ha figli
552         int simplified = 0; //E' 0 se non ho effettuato semplificazioni (valore inizializzato)
553
554         if (strcmp(t->nodeName, "and") == 0) { //Se il nodo attuale è un "and"
555             for(int i=0; i<t->nodesLen; i++) { //E se scorrendo tra i figli ...
556                 if (strcmp(t->nodes[i]->nodeName, "false") == 0) { //... trovo un "false"
557                     simplified = 1; //Mi segno che ho effettuato una semplificazione
558
559                     for (int j=0; j<t->nodesLen; j++) //Dealloco tutto ...
560                         recFree(t->nodes[j]);
561

```

```

562         strcpy(t->nodeName, "false"); //... e il nodo "and" diventa un "false"
563         t->nodesLen = 0;
564         break;
565     }
566 }
567 }
568
569 if (strcmp(t->nodeName, "or") == 0) { //Se il nodo attuale è un "or"
570     for(int i=0; i<t->nodesLen; i++) { //E se scorrendo tra i figli ...
571         if (strcmp(t->nodes[i]->nodeName, "true") == 0) { //... trovo un "true"
572             simplified = 1; //Mi segno che ho effettuato una semplificazione
573
574             for (int j=0; j<t->nodesLen; j++) //Dealloco tutto ...
575                 recFree(t->nodes[j]);
576
577             strcpy(t->nodeName, "true"); //... e il nodo "or" diventa un "true"
578             t->nodesLen = 0;
579             break;
580         }
581     }
582 }
583
584 if (!simplified) //Se non ho effettuato semplificazioni ...
585     for(int i=0; i<t->nodesLen; i++)
586         simplify(t->nodes[i]); //... tento di semplificare i figli del nodo attuale
587 }
588 }

```

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`.

4.2.7 Funzioni gcd e lcm

```

14 long int gcd(long int a, long int b) { //Massimo comun divisore
15     return b == 0 ? a : gcd(b, a % b);
16 }
17
18
19 long int lcm(long int a, long int b) { //Minimo comune multiplo
20     return abs((a / gcd(a, b)) * b);
21 }

```

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 Euclide⁷ 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.

4.2.8 Funzione `getLCM`

```

179 int getLCM(t_syntaxTree* tree, char* var) { //Restituisce il minimo comune multiplo dei
180                                           //coefficienti di var che appaiono in tutto
181                                           //l'albero
182     if (tree->nodeName[0] == '*') { //Se sono in un nodo "*" e ...
183       if (strcmp(((t_syntaxTree *)tree->nodes[1])->nodeName, var) == 0) {
184         //... se il secondo figlio è var (non può mai essere il primo!)
185         return atoi(((t_syntaxTree *) tree->nodes[0])->nodeName); //ritorno il coefficiente
186       }
187     }
188
189     int l = 1; //Inizializzo a 1 nel caso il nodo attuale sia senza figli
190
191     for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli e ...
192       l = lcm(l, getLCM((t_syntaxTree *) tree->nodes[i], var)); //... scarico ricorsivamente
193     }
194
195     return l;
196 }

```

`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 correttamente 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:

- 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,⁸ 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

⁷Euclid. *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.

⁸ghilardi, *mcmt: model checker modulo theories*, cit.

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 $(+ (* c1 x1) (* c2 x2) \dots (* c3 x3))$, 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.

4.2.9 Funzione normalize

```
199 void normalize(t_syntaxTree* tree, char* var) { //Modifica i coefficienti della variabili
200                                             //trasformandoli in 1 o -1
201     int lcm = getLCM(tree, var);
202     int c = lcm;
203
204     for (int i=0; i<tree->nodesLen; i++) { //Scorro tra i figli della radice
205         if (strcmp("=", tree->nodes[i]->nodeName) == 0 || //Se sono in un "=" o ...
206             strcmp("div", tree->nodes[i]->nodeName) == 0) { //... in un "div"
207             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
208
209             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro tra gli addendi
210                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
211                     c = atoi(addends[j]->nodes[0]->nodeName); //c è il coefficiente della variabile
212             }
213
214             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro fra gli addendi
215                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) { //Se sono sulla variabile
216                     strcpy(addends[j]->nodeName, var); //In questo modo il coefficiente è 1
217                     free(addends[j]->nodes[0]);
```



```

218         free(addends[j]->nodes[1]);
219         addends[j]->nodesLen = 0;
220     }
221     else { //Se non sono sulla variabile
222         sprintf(addends[j]->nodes[0]->nodeName,
223             "%d",
224             atoi(addends[j]->nodes[0]->nodeName)*lcm/c); //Ricalcolo i coefficienti
225     }
226 }
227
228 //Ricalcolo anche i coefficienti del termine costante a secondo membro di "=" o "div"
229 sprintf(tree->nodes[i]->nodes[1]->nodeName,
230     "%d",
231     atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/c);
232 }
233
234 else if (strcmp(">", tree->nodes[i]->nodeName) == 0) { //Se invece sono su un ">"
235     t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
236
237     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
238         if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
239             c = atoi(addends[j]->nodes[0]->nodeName); //Mi segno il coefficiente di var
240         }
241     }
242
243     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
244         if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) { //Se sono sulla variabile
245             if (c>0) strcpy(addends[j]->nodeName, ""); //Se il coefficiente è positivo non
246                 //faccio niente
247             else strcpy(addends[j]->nodeName, "-"); //Altrimenti lo cambio di segno
248             strcat(addends[j]->nodeName, var);
249             free(addends[j]->nodes[0]);
250             free(addends[j]->nodes[1]);
251             addends[j]->nodesLen = 0;
252         }
253         else { //Se non sono sulla variabile
254             sprintf(addends[j]->nodes[0]->nodeName,
255                 "%d",
256                 atoi(addends[j]->nodes[0]->nodeName)*abs(lcm/c)); //Ricalcolo i
257                 //coefficienti
258         }
259     }
260
261     //Ricalcolo anche i coefficienti del secondo membro di ">"
262     sprintf(tree->nodes[i]->nodes[1]->nodeName,
263         "%d",
264         atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/abs(c));
265 }
266 }
267

```

```

268     tree->nodesLen++;
269     tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
270     tree->nodes[tree->nodesLen-1] = malloc(sizeof(t_syntaxTree));
271     strcpy(tree->nodes[tree->nodesLen-1]->nodeName, "div");
272     tree->nodes[tree->nodesLen-1]->nodesLen = 2;
273     tree->nodes[tree->nodesLen-1]->nodes = malloc(sizeof(t_syntaxTree*) * 2);
274     tree->nodes[tree->nodesLen-1]->nodes[0] = malloc(sizeof(t_syntaxTree));
275     tree->nodes[tree->nodesLen-1]->nodes[1] = malloc(sizeof(t_syntaxTree));
276     tree->nodes[tree->nodesLen-1]->nodes[0]->nodesLen = 0;
277     tree->nodes[tree->nodesLen-1]->nodes[0]->nodes = NULL;
278     tree->nodes[tree->nodesLen-1]->nodes[1]->nodesLen = 0;
279     tree->nodes[tree->nodesLen-1]->nodes[1]->nodes = NULL;
280     strcpy(tree->nodes[tree->nodesLen-1]->nodes[0]->nodeName, var);
281     sprintf(tree->nodes[tree->nodesLen-1]->nodes[1]->nodeName, "%d", lcm);
282 }

```

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

4.2.10 Funzione `minInf`

```

315 t_syntaxTree* minInf(t_syntaxTree* tree, char* var) { //Calcola  $\varphi_{-\infty}$ 
316     t_syntaxTree* nTree = recCopy(tree); //La funzione lavora su una copia dell'albero
317
318     char minvar[16]; //La variabile con "-" davanti
319     minvar[0] = '\0';
320     strcpy(minvar, "-");
321     strcat(minvar, var);
322
323     for (int i=0; i<nTree->nodesLen; i++) { //Scorro sui figli della radice
324         if (strcmp(">", nTree->nodes[i]->nodeName) == 0) { //Se sono in un ">"
325             t_syntaxTree** addends = nTree->nodes[i]->nodes[0]->nodes;
326
327             for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
328                 if (strcmp(addends[j]->nodeName, var) == 0) //Se l'addendo contiene la variabile
329                     strcpy(nTree->nodes[i]->nodeName, "false"); //L'addendo diventa "false"
330                 else if (strcmp(addends[j]->nodeName, minvar) == 0) //Se l'addendo contiene la
331                                                             //variabile col segno "-"
332                     strcpy(nTree->nodes[i]->nodeName, "true"); //L'addendo diventa "true"
333             }
334
335             for (int j=0; j<nTree->nodes[i]->nodesLen; j++) //Scorro sui figli dei figli della
336                                                         //radice
337
338                 recFree(nTree->nodes[i]->nodes[j]); //E dealloco tutto
339
340                 free(nTree->nodes[i]->nodes);
341                 nTree->nodes[i]->nodesLen = 0;
342                 nTree->nodes[i]->nodes = NULL;
343         }
344     }

```

```

345     else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) { //Se sono in un "="
346         for (int j=0; j<nTree->nodes[i]->nodesLen; j++) //Scorro sui figli ...
347             recFree(nTree->nodes[i]->nodes[j]); //... e dealloco tutto
348
349         free(nTree->nodes[i]->nodes);
350         nTree->nodes[i]->nodesLen = 0;
351         nTree->nodes[i]->nodes = NULL;
352         strcpy(nTree->nodes[i]->nodeName, "false"); //I nodi possono diventare solo "false"
353     }
354 }
355
356 return nTree;
357 }

```

Prima di passare alla discussione della funzione `newFormula`, che effettivamente 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 già discusso.

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

4.2.11 Funzione calcm

```

386 int calcm(t_syntaxTree* tree, char* var) { //Calcolo il minimo comune multiplo di tutti i
387                                         //coefficienti della variabile
388     int m=1;
389
390     for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli della radice
391         if(strcmp(tree->nodes[i]->nodeName, "div") == 0) { //Se sono in un "div"
392
393             if(strcmp(tree->nodes[i]->nodes[0]->nodeName, var) == 0) //Se trovo la variabile senza
394                                                                     //coefficiente (i.e. 1 o -1)
395                 m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName)); //Calcolo il m.c.m.
396
397             else if(strcmp(tree->nodes[i]->nodes[0]->nodeName, "+") == 0) { //Altrimenti se trovo
398                                                                     //un "+"
399                 for(int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro tra gli addendi
400                     if (strcmp(tree->nodes[i]->nodes[0]->nodes[j]->nodeName, var) == 0) {
401                         m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
402                         break;
403                     }
404                 }
405             }
406         }
407     }
408
409     return m;
410 }

```

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.

4.2.12 Funzione boundaryPoints

```

413 t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var) { //Calcolo dei boundary points
414     char str[16];
415     str[0] = '\0';
416     t_syntaxTree* bPoints = malloc(sizeof(t_syntaxTree)); //Salvo i boundary points in un
417                                                         //albero
418     bPoints->nodes = NULL;
419     strcpy(bPoints->nodeName, "bPoints"); //Nome utile solo per il debugging
420     bPoints->nodesLen = 0;
421
422     for(int i=0; i<tree->nodesLen; i++) { //Scorro sui figli della radice
423         if (strcmp(tree->nodes[i]->nodeName, "=") == 0) { //Se sono in un "="
424             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
425
426             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) { //Scorro sugli addendi
427                 if (strcmp(var, addends[j]->nodeName) == 0) { //Se trovo la variabile
428                     bPoints->nodesLen++; //Aggiungo un boundary point
429                     bPoints->nodes = realloc(bPoints->nodes,
430                                             sizeof(t_syntaxTree *) * bPoints->nodesLen);
431                     t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
432                     bp->nodes = NULL;
433                     strcpy(bp->nodeName, "+");
434                     bp->nodesLen = 0;
435
436                     for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) { //Scorro sugli addendi
437                         if (strcmp(var, addends[k]->nodeName) != 0) { //Se non sono sulla variabile
438                             bp->nodesLen++; //Aggiungo comunque
439                             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
440                             bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
441
442                             sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
443                             strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
444                         }
445                     }
446
447                     bp->nodesLen++;
448                     bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
449                     bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
450                     bp->nodes[bp->nodesLen-1]->nodesLen = 0;
451                     bp->nodes[bp->nodesLen-1]->nodes = NULL;
452                     sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
453                     strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
454

```

```

455         bPoints->nodes[bPoints->nodesLen-1] = bp;
456         break;
457     }
458 }
459 }
460
461 if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
462     t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
463
464     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
465         if (strcmp(var, addends[j]->nodeName) == 0) {
466             bPoints->nodesLen++;
467             bPoints->nodes = realloc(bPoints->nodes,
468                                     sizeof(t_syntaxTree *) * bPoints->nodesLen);
469             t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
470             bp->nodes = NULL;
471             strcpy(bp->nodeName, "+");
472             bp->nodesLen = 0;
473
474             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
475                 if (strcmp(var, addends[k]->nodeName) != 0) {
476                     bp->nodesLen++;
477                     bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
478                     bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
479                     sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
480                     strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
481                 }
482             }
483
484             bp->nodesLen++;
485             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
486             bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
487             bp->nodes[bp->nodesLen-1]->nodesLen = 0;
488             bp->nodes[bp->nodesLen-1]->nodes = NULL;
489             sprintf(str, "%d", +atoi(tree->nodes[i]->nodes[1]->nodeName));
490             strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
491
492             bPoints->nodes[bPoints->nodesLen-1] = bp;
493             break;
494         }
495     }
496 }
497 }
498
499 return bPoints;
500 }

```

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 formule φ' e

$\varphi'_{-\infty}$ e la variabile da eliminare, è al suo interno che vengono effettuate le chiamate a `boundaryPoints` e `calcm`.

4.2.13 Funzione newFormula

```

504 t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
505     int m = calcm(minf, var); //Minimo comune multiplo dei coefficienti di var
506     t_syntaxTree* val;
507     char str[16];
508     t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree)); //Nuovo albero che verrà restituito
509     strcpy(nTree->nodeName, "or"); //La radice del nuovo albero è un "or"
510     nTree->nodesLen = 0;
511     nTree->nodes = NULL;
512
513     t_syntaxTree* t;
514     t_syntaxTree* bp;
515     t_syntaxTree *bPts = boundaryPoints(tree, var); //Calcola i boundary points
516
517     //La seguente è semplicemente una applicazione della formula del teorema
518     for(int i=1; i<=m; i++) {
519         nTree->nodesLen++;
520         nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
521         t = recCopy(minf);
522         val = malloc(sizeof(t_syntaxTree));
523         sprintf(str, "%d", i);
524         strcpy(val->nodeName, str);
525         val->nodesLen = 0;
526         val->nodes = NULL;
527         eval(t, var, val);
528         recFree(val);
529         nTree->nodes[nTree->nodesLen-1] = t;
530
531         for(int j=0; j<bPts->nodesLen; j++) {
532             nTree->nodesLen++;
533             nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
534             t = recCopy(tree);
535             bp = recCopy(bPts->nodes[j]);
536             sprintf(str, "%d", i+atoi(bp->nodes[bp->nodesLen-1]->nodeName));
537             strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
538             eval(t, var, bp);
539             recFree(bp);
540
541             nTree->nodes[nTree->nodesLen-1] = t;
542         }
543     }
544
545     recFree(bPts); //Non servono più, dealloco
546     return nTree;
547 }
```

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.

4.2.14 Funzione eval

```

361 void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val) {
362     for (int i=0; i<tree->nodesLen; i++) { //Scorro sui figli di tree
363         if (strcmp(tree->nodes[i]->nodeName, var) == 0) { //Nel caso trovi var
364             recFree(tree->nodes[i]); //Dealloco e ...
365             tree->nodes[i] = recCopy(val); //... sostituisco con una copia di val
366         }
367     else { //Nel caso non trovi var potrei comunque ancora trovare var con "-" davanti
368         char mvar[17] = "-"; //E' var con "-" davanti
369         strcat(mvar, var);
370         if (strcmp(tree->nodes[i]->nodeName, mvar) == 0) { //Se trovo var con "-" davanti
371             recFree(tree->nodes[i]); //Dealloco tutto
372             //Creo un nuovo nodo "-" contenente val come unico figlio
373             tree->nodes[i] = malloc(sizeof(t_syntaxTree));
374             strcpy(tree->nodes[i]->nodeName, "-");
375             tree->nodes[i]->nodesLen = 1;
376             tree->nodes[i]->nodes = malloc(sizeof(t_syntaxTree*));
377             tree->nodes[i]->nodes[0] = recCopy(val);
378         }
379     }
380
381     eval(tree->nodes[i], var, val); //Scarico ricorsivamente su tutti i figli
382 }
383 }
```

5 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.

5.1 Programmi di esempio

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "cooper.h"
4
5  int main(int argc, char** argv) {
6      char* str;
7
8      if (argc == 3) {
9          str = cooperToStr(argv[1], argv[2]);
10         printf("%s", str);
11     }
12     else
13         printf("Numero errato di argomenti!\n");
14
15     free(str);
16
17     return 0;
18 }
```

Il secondo programma di esempio è `test2.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "cooper.h"
4
5  int main(int argc, char** argv) {
6      char** array;
7      int len;
8
9      if (argc == 3) {
10         array = cooperToArray(argv[1], argv[2], &len);
11
12         for (int i=0; i<len; i++) {
13             printf("%s\n", array[i]);
14             free(array[i]);
15         }
16
17         free(array);
18     }
19     else
20         printf("Numero errato di argomenti!\n");
```



```

21
22     return 0;
23 }

```

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

```

677 char** cooperToArray(char* wff, char* var, int* len) {
678     t_syntaxTree* tree, *minf, *f;
679     char* buffer;
680     char** array;
681
682     tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
683     normalize(tree, var); //Trasforma l'albero di tree
684     minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
685     f = newFormula(tree, minf, var); //Restituisce la formula equivalente
686     simplify(f); //opzionale
687     adjustForYices(f);
688
689     *len = f->nodesLen;
690
691     array = malloc(sizeof(char*) * *len);
692
693     for (int i=0; i<*len; i++) {
694         buffer = treeToStr(f->nodes[i]);
695         array[i] = malloc(sizeof(char) * strlen(buffer));
696         strcpy(array[i], buffer);
697         free(buffer);
698     }
699
700     recFree(tree); //Libera la memoria
701     recFree(minf);
702     recFree(f);
703
704     return array;
705 }

```

5.2 Il Makefile

```

1 SHELL := /bin/bash
2 PARAMS = -std=c99 -O3 -Wall -g #compila nello standard C99 e abilita tutti i warning
3 leak-check = yes #valgrind effettua una ricerca dei leak più accurata
4 track-origins = yes #valgrind fornisce più informazioni
5 wff = "(and (= (+ (* -2 x) (* 3 y)) 3) \
6           (> (+ (* 5 x) (* 3 y)) 1) \
7           (div (+ (* 2 x) (* 4 y)) 1))" #formula in ingresso
8 wff = "(and (div (+ (* 3 z)) 3) (= (+ (* 2 y) (* 3 x)) 2) (= (+ (* 2 x)) 4))"
9 wff= "(and (> (+ (* 1 x)) 5) (> (+ (* -1 x) (* 1 y)) 0) (div (+ (* 1 1) (* 2 x)) 3))"
10 #wff="(and (= (+ (* 2 a) (* 3 b) (* 4 c)) 3) (> (+ (* 3 x) (* 2 y)) 1) (= (+ (* 2 x) (* 4 y)) 3) (>
11 #wff="(and (= (+ (* 2 x) (* -1 y)) 3) (= (+ (* 4 a) (* 1 y)) 1) (> (+ (* -2 x) (* 1 y)) -10) (= (+
12 vars = "x y" #variabili presenti nella formula
13 var = "x" #variabile da eliminare

```

```

14
15 test: test.c cooper.o
16     gcc $(PARAMS) test.c cooper.o -o test
17
18 test2: test2.c cooper.o
19     gcc $(PARAMS) test2.c cooper.o -o test2
20
21 cooper.o: cooper.c cooper.h
22     gcc $(PARAMS) -c cooper.c -o cooper.o
23
24 run: test #esegue test e restituisce il tempo impiegato
25     @echo -e 'Elimino la variabile $(var) dalla seguente formula:\n$(wff) ---> \n'
26     @time ./test $(wff) $(var)
27
28 run2: test2
29     @time ./test2 $(wff) $(var)
30
31 eq: test eq.py #prende la formula e la variabile da eliminare, la elimina controlla con z3 se il r
32     @python3 eq.py "$(formula)" "$(variables)" "$(guess)" #la prima in variables è quella da e
33
34 valgrind: test
35     valgrind --track-origins=$(track-origins) \
36         --leak-check=$(leak-check) ./test $(wff) $(var)
37
38 valgrind2: test2
39     valgrind --track-origins=$(track-origins) \
40         --leak-check=$(leak-check) ./test2 $(wff) $(var)
41
42 debug: test #esegue test col debugger gdb
43     gdb --args test $(wff) $(var)
44
45 eval: test3 #valuta il valore della formula equivalente,
46     #funziona solo se ogni variabile è già stata eliminata
47     ./eval.scm "`./test3 $(wff) $(vars) | tail -n 1`"
48
49 clean:
50     rm -f *.o
51     rm -f test test2
52     rm -f eq.smt

```

Si consideri ora il seguente makefile: È 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.⁹

5.3 Valutazione ed equivalenza

```

1  #!/bin/guile \N
2  -e main -s

```

⁹GNU. *GNU Ubiquitous Intelligent Language for Extensions (GUILE)*.

```

3  !#
4
5  (use-modules (ice-9 format) (ice-9 eval-string))
6
7  (define (div a b)
8    (if (= (remainder a b) 0) #t #f))
9
10 (define true #t)
11
12 (define false #f)
13
14 (define (main args)
15   (let ((str (cadr args)))
16     (format #t
17       "\nInput: ~s\nEvaluated: ~s\n"
18       str
19       (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 soddisfacibilità si usi invece la regola `eq` del `makefile`. Tale regola esegue il seguente script Python.¹⁰

```

1  #!/usr/bin/env python3
2  from sys import argv
3  from subprocess import run, PIPE
4
5
6  def main():
7      if len(argv) != 4:
8          print("Wrong arguments number!")
9      else:
10         formula = argv[1]
11         variables = argv[2].split() #tutte le variabili
12         var = variables[0] #la prima è quella da eliminare
13         guess = argv[3]
14         smt_source = ""
15
16         for v in variables:
17             if v is not var:
18                 smt_source += "(declare-const {} Int)\n".format(v)
19
20         wff_out = run(["./test", formula, var], stdout=PIPE).stdout.decode()
21         smt_source += "(assert (not (= {} {})))\n".format(wff_out, guess)
22         smt_source += "(check-sat)\n"
23
24         with open("eq.smt2", "w") as source:

```

¹⁰Python Software Foundation. *Python language*. Ver. 3.7.2. 2019. URL: <https://www.python.org/>.

```

25         print(smt_source, file=source)
26
27     print("Verifico se {} e' equivalente a \n{}".format(guess, wff_out))
28     print("Eseguo con z3 il file 'eq.smt2' contenente: \n{}".format(smt_source))
29
30     result = run(["z3", "eq.smt2"], stdout=PIPE).stdout.decode()
31
32     if "unsat" in result:
33         print("Sono equivalenti")
34     elif "error" in result:
35         print("Errore di z3:\n{}".format(result))
36     elif "sat" in result:
37         print("Non sono equivalenti")
38
39
40 if __name__ == '__main__':
41     main()

```

Tale script genera un opportuno sorgente per z3¹¹ 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.

5.4 Esempi

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

5.4.1 Esempio 1

Si voglia eliminare la variabile x dalla seguente formula:

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

È 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)"

```

¹¹Microsoft 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 però il primo figlio deve essere necessariamente 1 o -1. Questo si rivela fondamentale per inserire costanti nelle formule con `div`

5.4.2 Esempio 2

É anche possibile verificare direttamente la verità 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, può 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.

5.4.3 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

1	Aritmetica di Presburger	1
2	L'algoritmo di Cooper	1
2.1	Processo di semplificazione	1
2.2	Normalizzazione dei coefficienti	2
2.3	Costruzione di $\varphi'_{-\infty}$	2
2.4	Calcolo dei boundary points	2
2.5	Eliminazione dei quantificatori	3
3	Complessità computazionale	4
3.1	Formalizzazione dell'aritmetica di Presburger	4
3.2	L'algoritmo di Cooper come procedura decisionale	4
3.3	Analisi e stima della complessità	5
4	Implementazione	8
4.1	Struttura e design	8
4.2	Analisi del codice	9
4.2.1	Funzione <code>cooperToStr</code>	9
4.2.2	Segnatura di <code>t_syntaxTree</code>	9
4.2.3	Funzione <code>recFree</code>	10
4.2.4	Funzione <code>parse</code>	11
4.2.5	Funzione <code>treeToStr</code>	12
4.2.6	Funzione <code>simplify</code>	13
4.2.7	Funzioni <code>gcd</code> e <code>lcm</code>	14
4.2.8	Funzione <code>getLCM</code>	15
4.2.9	Funzione <code>normalize</code>	16
4.2.10	Funzione <code>minInf</code>	18
4.2.11	Funzione <code>calcm</code>	19
4.2.12	Funzione <code>boundaryPoints</code>	20
4.2.13	Funzione <code>newFormula</code>	22
4.2.14	Funzione <code>eval</code>	23
5	Utilizzo	24
5.1	Programmi di esempio	24
5.2	Il Makefile	25
5.3	Valutazione ed equivalenza	26
5.4	Esempi	28
5.4.1	Esempio 1	28
5.4.2	Esempio 2	29
5.4.3	Esempio 3	29

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 (GUILF)*.
- 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/>.
- 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. URL: <https://doi-org.pros.lib.unimi.it:2050/10.1080/014453409108837187>.
- Python Software Foundation. *Python language*. Ver. 3.7.2. 2019. URL: <https://www.python.org/>.