

Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

27 marzo 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 diseguaglianze 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 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:

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

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 che non siano $<$, $|$ or \dagger 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. $\epsilon_i \dagger x + d_i$

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

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

3.3 Analisi e stima della complessità

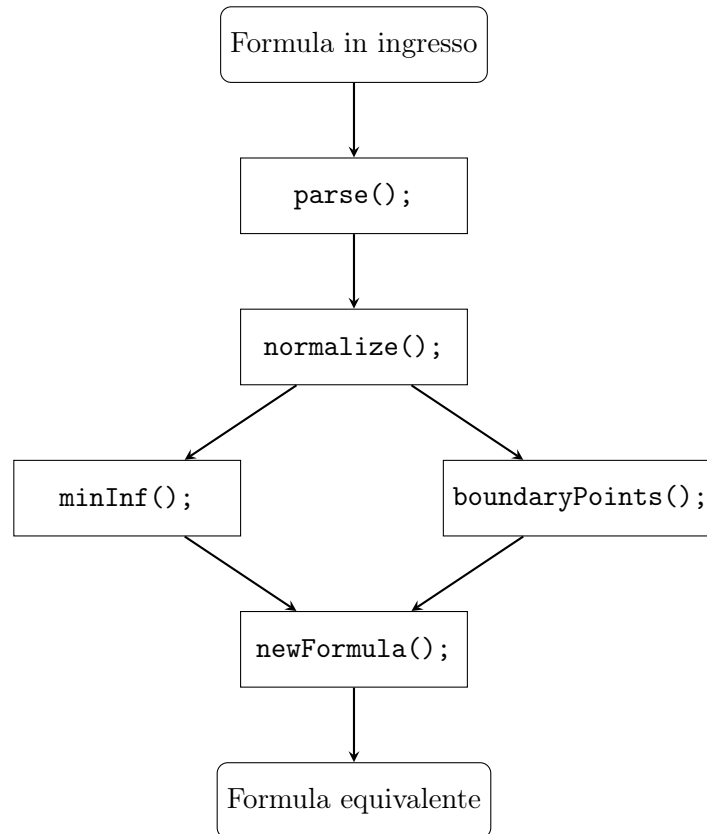
Theorem 3.1 (Pythagorean theorem). *This is a theorem about right triangles and can be summarised in the next equation*

$$x^2 + y^2 = z^2$$

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.

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)`.

⁵c99.

⁶smtlib.

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 della 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

```

609 char* cooperToStr(char* wff, char* var) {
610     t_syntaxTree* tree, *minf, *f;
611     char* str;
612
613     tree = parse(wff, 1); //Genera l'albero sintattico a partire dalla stringa
614     normalize(tree, var); //Trasforma l'albero di tree
615     //printf("\nNormalizzato%s\n\n", treeToStr(tree));
616     minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
617     f = newFormula(tree, minf, var); //Restituisce la formula equivalente
618     simplify(f); //opzionale
619     adjustForYices(f);
620     str = treeToStr(f); //Genera la stringa a partire dall'albero
621
622     recFree(tree); //Libera la memoria
623     recFree(minf);
624     recFree(f);
625
626     return str;
627 }
```

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

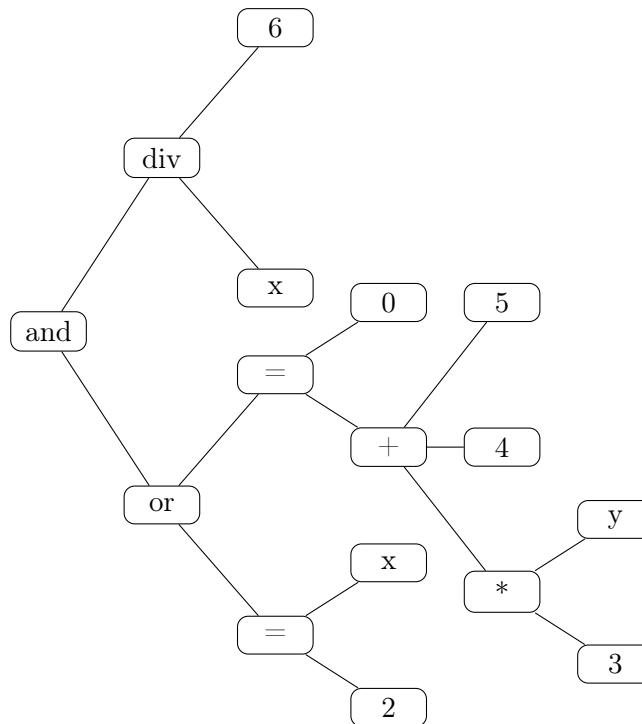
```

18 typedef struct t_syntaxTree {
19     char nodeName[16];
20     int nodesLen;
21     struct t_syntaxTree** nodes;
22 } 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

```

272 void recFree(t_syntaxTree* tree) {
273     for (int i=0; i<tree->nodesLen; i++) {
274         recFree(tree->nodes[i]);
275     }
276
277     free(tree->nodes);
278     free(tree);
279 }

```

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

```
109 t_syntaxTree* parse(char* wff, int strict) {
110     char* wffSpaced = malloc(sizeof(char));
111     wffSpaced[0] = wff[0];
112     int j = 1;
113
114     for (int i = 1; i < strlen(wff) + 1; i++) {
115
116         if (wff[i - 1] == '(') {
117             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
118             wffSpaced[j] = ' ';
119             wffSpaced[j + 1] = wff[i];
120             j += 2;
121         }
122
123         else if (wff[i + 1] == ')') {
124             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
125             wffSpaced[j] = wff[i];
126             wffSpaced[j + 1] = ' ';
127             j += 2;
128         }
129
130         else {
131             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 1));
132             wffSpaced[j] = wff[i];
133             j++;
134         }
135     }
136
137     char* token;
138     int nTokens = 1;
139     char** tokens = malloc(sizeof(char *));
140     tokens[0] = strtok(wffSpaced, " ");
141
142     while ((token = strtok(NULL, " ")) != NULL) {
143         nTokens++;
144         tokens = realloc(tokens, sizeof(char *) * nTokens);
145         tokens[nTokens - 1] = token;
146     }
147
148     int countPar = 0;
149
150     for(int i=0; i<nTokens; i++) {
151         for(int j=0; j<strlen(tokens[i]); j++)
152             if(tokens[i][j] == ')' && j!= 0)
153                 ERROR("Parsing error: every S-expression must \
154 have a root and at least an argument");
155         if (tokens[i][0] == '(') countPar++;
156         if (tokens[i][0] == ')') countPar--;
```

```

157     }
158
159     if (countPar != 0)
160         ERROR("Parsing error: the number of parentheses is not even");
161
162     t_syntaxTree* syntaxTree = buildTree(0, tokens);
163
164     if (strict) checkTree(syntaxTree); //chiama exit() se l'albero non va bene
165
166     free(wffSpaced);
167     free(tokens);
168
169     return syntaxTree;
170 }

```

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`

```

581 char* treeToStr(t_syntaxTree* tree) {
582     char* str=malloc(sizeof(char));
583     str[0] = '\0';
584     recTreeToStr(tree, &str, 1);
585     return str;
586 }

```

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.

```

551 int recTreeToStr(t_syntaxTree* t, char** str, int len) {
552     if (t->nodesLen == 0) {
553         int nLen = len + strlen(t->nodeName);
554         *str = realloc(*str, sizeof(char) * nLen);
555         strcat(*str, t->nodeName);
556         return nLen;
557     }
558
559     else {
560         int nLen = len + strlen(t->nodeName) + 1;
561         *str = realloc(*str, sizeof(char) * nLen);
562         strcat(*str, "(");
563         strcat(*str, t->nodeName);
564
565         for (int i=0; i<t->nodesLen; i++) {
566             nLen++;

```

```

567     *str = realloc(*str, sizeof(char) * nLen);
568     strcat(*str, " ");
569     nLen = recTreeToStr(t->nodes[i], str, nLen);
570 }
571
572 nLen++; //nLen++;
573 *str = realloc(*str, sizeof(char) * nLen);
574 strcat(*str, ")");
575
576 return nLen;
577 }
578 }

```

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.

```

525     }
526 }
527 }
528
529 if (strcmp(t->nodeName, "or") == 0) {
530     for(int i=0; i<t->nodesLen; i++) {

```

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`

```

510 void simplify(t_syntaxTree* t) {
511     if (t->nodesLen != 0) {
512         int simplified = 0;
513
514         if (strcmp(t->nodeName, "and") == 0) {
515             for(int i=0; i<t->nodesLen; i++) {
516                 if (strcmp(t->nodes[i]->nodeName, "false") == 0) {
517                     simplified = 1;
518
519                     for (int j=0; j<t->nodesLen; j++)
520                         recFree(t->nodes[j]);
521
522                     strcpy(t->nodeName, "false");
523                     t->nodesLen = 0;
524                     break;
525                 }
526             }
527         }
528
529         if (strcmp(t->nodeName, "or") == 0) {
530             for(int i=0; i<t->nodesLen; i++) {
531                 if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
532                     simplified = 1;

```

```

533
534     for (int j=0; j<t->nodesLen; j++)
535         recFree(t->nodes[j]);
536
537     strcpy(t->nodeName, "true");
538     t->nodesLen = 0;
539     break;
540 }
541 }
542 }
543
544 if (!simplified)
545     for(int i=0; i<t->nodesLen; i++)
546         simplify(t->nodes[i]);
547 }
548 }

```

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 infine 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

```

8  long int gcd(long int a, long int b) {
9      return b == 0 ? a : gcd(b, a % b);
10 }

13 long int lcm(long int a, long int b) {
14     return abs((a / gcd(a, b)) * b);
15 }

```

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.

⁷euclid.

4.2.8 Funzione getLCM

```
173 int getLCM(t_syntaxTree* tree, char* var) {
174     if (tree->nodeName[0] == '*') {
175         if (strcmp(((t_syntaxTree *)tree->nodes[1])->nodeName, var) == 0) {
176             return atoi(((t_syntaxTree *) tree->nodes[0])->nodeName);
177         }
178     }
179
180     int l = 1;
181
182     for(int i=0; i<tree->nodesLen; i++) {
183         l = lcm(l, getLCM((t_syntaxTree *) tree->nodes[i], var));
184     }
185
186     return l;
187 }
```

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 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 reso 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:

⁸Ghilardi, *MCMT: Model Checker Modulo Theories*, cit.

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

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

```
190 void normalize(t_syntaxTree* tree, char* var) {
191     int lcm = getLCM(tree, var);
192     int c = 1;
193
194     for (int i=0; i<tree->nodesLen; i++) {
195         if (strcmp("=", tree->nodes[i]->nodeName) == 0 ||
196             strcmp("div", tree->nodes[i]->nodeName) == 0) {
197             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
198
199             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
200                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
201                     c = atoi(addends[j]->nodes[0]->nodeName);
202             }
203
204
205             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
206                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
207                     strcpy(addends[j]->nodeName, var);
208                     free(addends[j]->nodes[0]);
209                     free(addends[j]->nodes[1]);
210                     addends[j]->nodesLen = 0;
211                 }
212                 else {
213                     sprintf(addends[j]->nodes[0]->nodeName,
214                             "%d",
215                             atoi(addends[j]->nodes[0]->nodeName)*lcm/c);
216                 }
217             }
218
219             sprintf(tree->nodes[i]->nodes[1]->nodeName,
220                     "%d",
221                     atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/c);
```

```

222     }
223
224     else if (strcmp(">", tree->nodes[i]->nodeName) == 0) {
225         t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
226
227         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
228             if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
229                 c = atoi(addends[j]->nodes[0]->nodeName);
230         }
231
232
233         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
234             if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
235                 if(c > 0) strcpy(addends[j]->nodeName, "");
236                 else strcpy(addends[j]->nodeName, "-");
237                 strcat(addends[j]->nodeName, var);
238                 free(addends[j]->nodes[0]);
239                 free(addends[j]->nodes[1]);
240                 addends[j]->nodesLen = 0;
241             }
242             else {
243                 sprintf(addends[j]->nodes[0]->nodeName,
244                         "%d",
245                         atoi(addends[j]->nodes[0]->nodeName)*lcm/abs(c));
246             }
247         }
248
249         sprintf(tree->nodes[i]->nodes[1]->nodeName,
250                 "%d",
251                 atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/abs(c));
252     }
253 }
254
255 tree->nodesLen++;
256 tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
257 tree->nodes[tree->nodesLen-1] = malloc(sizeof(t_syntaxTree));
258 strcpy(tree->nodes[tree->nodesLen-1]->nodeName, "div");
259 tree->nodes[tree->nodesLen-1]->nodesLen = 2;
260 tree->nodes[tree->nodesLen-1]->nodes = malloc(sizeof(t_syntaxTree*) * 2);
261 tree->nodes[tree->nodesLen-1]->nodes[0] = malloc(sizeof(t_syntaxTree));
262 tree->nodes[tree->nodesLen-1]->nodes[1] = malloc(sizeof(t_syntaxTree));
263 tree->nodes[tree->nodesLen-1]->nodes[0]->nodesLen = 0;
264 tree->nodes[tree->nodesLen-1]->nodes[0]->nodes = NULL;
265 tree->nodes[tree->nodesLen-1]->nodes[1]->nodesLen = 0;
266 tree->nodes[tree->nodesLen-1]->nodes[1]->nodes = NULL;
267 strcpy(tree->nodes[tree->nodesLen-1]->nodes[0]->nodeName, var);
268 sprintf(tree->nodes[tree->nodesLen-1]->nodes[1]->nodeName, "%d", lcm);
269 }

```

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

```

301 t_syntaxTree* minInf(t_syntaxTree* tree, char* var) {
302     t_syntaxTree* nTree = recCopy(tree);
303
304     char minvar[16];
305     minvar[0] = '\0';
306     strcpy(minvar, "-");
307     strcat(minvar, var);
308
309     for (int i=0; i<nTree->nodesLen; i++) {
310         if (strcmp(">", nTree->nodes[i]->nodeName) == 0) {
311             t_syntaxTree** addends = nTree->nodes[i]->nodes;
312
313             for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) {
314                 if (strcmp(addends[j]->nodeName, var) == 0)
315                     strcpy(nTree->nodes[i]->nodeName, "false");
316                 else if (strcmp(addends[j]->nodeName, minvar) == 0)
317                     strcpy(nTree->nodes[i]->nodeName, "true");
318             }
319
320             for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
321                 recFree(nTree->nodes[i]->nodes[j]);
322
323             free(nTree->nodes[i]->nodes);
324             nTree->nodes[i]->nodesLen = 0;
325             nTree->nodes[i]->nodes = NULL;
326         }
327
328         else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) {
329             for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
330                 recFree(nTree->nodes[i]->nodes[j]);
331
332             free(nTree->nodes[i]->nodes);
333             nTree->nodes[i]->nodesLen = 0;
334             nTree->nodes[i]->nodes = NULL;
335             strcpy(nTree->nodes[i]->nodeName, "false");
336         }
337     }
338
339     return nTree;
340 }

```

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 lcm

```

355 int lcm(t_syntaxTree* tree, char* var) {
356     int m=1;
357
358     for(int i=0; i<tree->nodesLen; i++) {
359         if(strcmp(tree->nodes[i]->nodeName, "div") == 0) {
360
361             if(strcmp(tree->nodes[i]->nodes[0]->nodeName, var) == 0)
362                 m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
363
364             else if(strcmp(tree->nodes[i]->nodes[0]->nodeName, "+") == 0) {
365                 for(int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
366                     if (strcmp(tree->nodes[i]->nodes[0]->nodes[j]->nodeName, var) == 0) {
367                         m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
368                         break;
369                     }
370                 }
371             }
372         }
373     }
374
375     return m;
376 }

```

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

```

379 t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var) {
380     char str[16];
381     str[0] = '\0';
382     t_syntaxTree* bPoints = malloc(sizeof(t_syntaxTree));
383     bPoints->nodes = NULL;
384     strcpy(bPoints->nodeName, "bPoints"); //solo per debugging
385     bPoints->nodesLen = 0;
386
387     for(int i=0; i<tree->nodesLen; i++) {
388         if (strcmp(tree->nodes[i]->nodeName, "=") == 0) {
389             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
390
391             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
392                 if (strcmp(var, addends[j]->nodeName) == 0) {

```

```

393     bPoints->nodesLen++;
394     bPoints->nodes = realloc(bPoints->nodes, sizeof(t_syntaxTree *) * bPoints->nodesLen);
395     t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
396     bp->nodes = NULL;
397     strcpy(bp->nodeName, "+");
398     bp->nodesLen = 0;
399
400     for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
401         if (strcmp(var, addends[k]->nodeName) != 0) {
402             bp->nodesLen++;
403             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
404             bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
405             sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
406             strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
407         }
408     }
409
410     bp->nodesLen++;
411     bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
412     bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
413     bp->nodes[bp->nodesLen-1]->nodesLen = 0;
414     bp->nodes[bp->nodesLen-1]->nodes = NULL;
415     sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
416     strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
417
418     bPoints->nodes[bPoints->nodesLen-1] = bp;
419     break;
420 }
421 }
422 }
423
424 if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
425     t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
426
427     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
428         if (strcmp(var, addends[j]->nodeName) == 0) {
429             bPoints->nodesLen++;
430             bPoints->nodes = realloc(bPoints->nodes, 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++) {
437                 if (strcmp(var, addends[k]->nodeName) != 0) {
438                     bp->nodesLen++;
439                     bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
440                     bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
441                     sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
442                     strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);

```

```

443     }
444 }
445
446 bp->nodesLen++;
447 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
448 bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
449 bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
450 bp->nodes[bp->nodesLen - 1]->nodes = NULL;
451 sprintf(str, "%d", +atoi(tree->nodes[i]->nodes[1]->nodeName));
452 strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);
453
454 bPoints->nodes[bPoints->nodesLen-1] = bp;
455 break;
456 }
457 }
458 }
459 }
460
461 return bPoints;
462 }

```

Si discuta ora la funzione che restituisce la formula equivalente che poi `cooper` 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

```

465 t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
466     int m = calcm(minf, var);
467     t_syntaxTree* val;
468     char str[16];
469     t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree));
470     strcpy(nTree->nodeName, "or");
471     nTree->nodesLen = 0;
472     nTree->nodes = NULL;
473
474     t_syntaxTree* t;
475     t_syntaxTree* bp;
476     t_syntaxTree *bPts = boundaryPoints(tree, var);
477
478     for(int i=1; i<=m; i++) {
479         nTree->nodesLen++;
480         nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
481         t = recCopy(minf);
482         val = malloc(sizeof(t_syntaxTree));
483         sprintf(str, "%d", i);
484         strcpy(val->nodeName, str);
485         val->nodesLen = 0;
486         val->nodes = NULL;
487         eval(t, var, val);

```

```

488     recFree(val);
489     nTree->nodes[nTree->nodesLen-1] = t;
490
491     for(int j=0; j<bPts->nodesLen; j++) {
492         nTree->nodesLen++;
493         nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
494         t = recCopy(tree);
495         bp = recCopy(bPts->nodes[j]);
496         sprintf(str, "%d", i+atoi(bp->nodes[bp->nodesLen-1]->nodeName));
497         strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
498         eval(t, var, bp);
499         recFree(bp);
500
501         nTree->nodes[nTree->nodesLen-1] = t;
502     }
503 }
504
505     recFree(bPts);
506     return nTree;
507 }

```

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 ovviamente 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

```

343 void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val) {
344     for (int i=0; i<tree->nodesLen; i++) {
345         if (strcmp(tree->nodes[i]->nodeName, var) == 0) {
346             recFree(tree->nodes[i]);
347             tree->nodes[i] = recCopy(val);
348         }
349         else
350             eval(tree->nodes[i], var, val);
351     }
352 }

```

5 Utilizzo

In questa sezione verranno forniti alcuni semplici esempi di utilizzo, innanzitutto si sottolinea come l'implementazione dell'algoritmo termini con la funzione `cooper`, 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 Il programma `test.c`

Si consideri il seguente programma di esempio contenuto in `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!");
14
15     free(str);
16
17     return 0;
18 }
```

Si consideri ora il seguente `makefile`:

5.2 Il Makefile

```
1  SHELL := /bin/bash
2  PARAMS = -std=c99 -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  vars = "x y z" #variabili presenti nella formula
10 var = "x" #variabile da eliminare
11
12 test: test.c cooper.o
13     gcc $(PARAMS) test.c cooper.o -o test
14
15 test2: test2.c cooper.o
16     gcc $(PARAMS) test2.c cooper.o -o test2
17
18 cooper.o: cooper.c cooper.h
```

```


19      gcc $(PARAMS) -c cooper.c -o cooper.o
20
21  run: test #esegue test e restituisce il tempo impiegato
22      @echo -e 'Elimino la variabile $(var) dalla seguente formula:\n$(wff) ---> \n'
23      @time ./test $(wff) $(var)
24
25  run2: test2
26      @time ./test2 $(wff) $(var)
27
28  sat: test sat.py #verifica la soddisfacibilità della formula generata grazie a yices
29      ./sat.py $(wff) $(vars) $(var)
30
31  valgrind: test
32      valgrind --track-origins=$(track-origins) \
33      --leak-check=$(leak-check) ./test $(wff) $(var)
34
35  valgrind2: test2
36      valgrind --track-origins=$(track-origins) \
37      --leak-check=$(leak-check) ./test2 $(wff) $(var)
38
39  debug: test #esegue test col debugger gdb
40      gdb --args test $(wff) $(var)
41
42  eval: test3 #valuta il valore della formula equivalente,
43      #funziona solo se ogni variabile è già stata eliminata
44      ./eval.scm "`./test3 $(wff) $(vars) | tail -n 1`"
45
46  clean:
47      rm -f *.o
48      rm -f test test2

```

È semplice immaginare cosa facciano le regole `run`, `valgrind`, `debug` e `clean`. Ci si soffermi ora su `eval` e `sat`. 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 e soddisfacibilità

```

1  #!/bin/guile 
2  -e main -s
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)

```

⁹guile.

```

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 utilizza condivida la stessa sintassi di 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 `sat` del `makefile`. Tale regola esegue il seguente script Python.¹⁰

```

1  #!/bin/python3
2  from sys import argv
3  from subprocess import run
4
5
6  def main():
7      if len(argv) != 4:
8          print("Wrong arguments number!")
9      else:
10         wff = argv[1]
11         variables = argv[2].split()
12         var = argv[3]
13         yices = ""
14
15         for v in variables:
16             if v is not var:
17                 yices += "(define {}::int)\n".format(v)
18
19         wff_out = run(["./test", wff, var],
20                       capture_output=True).stdout.decode()
21         yices += "(assert {})\n".format(wff_out)
22
23         with open("source.ys", "w") as source:
24             print(yices, file=source)
25
26         run(["yices", "source.ys"])
27
28
29  if __name__ == '__main__':
30      main()

```

Tale script genera un opportuno sorgente `source.ys` per Yices¹¹ e successivamente lo esegue, per esempio se la regola `make sat` esegue `./sat.py "(and (> (+ (* 2 x) (* 3 y)) 1))" "x y" "x"` allora viene generato il seguente `source.ys` che viene poi eseguito da Yices che restituisce la stringa `"sat"`.

¹⁰python.

¹¹yices.

```

(define x::int)
(define y::int)
(assert (or (and false (div 1 3))
            (and (> (+ (* 2 x) (+ (* -2 x) 2)) 1) (div (+ (* -2 x) 2) 3))
            (and false (div 2 3))
            (and (> (+ (* 2 x) (+ (* -2 x) 3)) 1) (div (+ (* -2 x) 3) 3))
            (and false (div 3 3))
            (and (> (+ (* 2 x) (+ (* -2 x) 4)) 1) (div (+ (* -2 x) 4) 3))))
(check)

```

Ovvero l'algoritmo trasforma correttamente una formula soddisfacibile (non è difficile trovare dei valori di x e y che soddisfino la formula iniziale) in una formula senza la variabile x che a sua volta Yices dice essere ancora soddisfacibile. Questo genere di verifiche ovviamente non garantiscono la corretta implementazione, ciononostante permettono di guadagnare una certa fiducia nella stessa.

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	6
4.1	Struttura e design	6
4.2	Analisi del codice	7
4.2.1	Funzione <code>cooperToStr</code>	7
4.2.2	Segnatura di <code>t_syntaxTree</code>	7
4.2.3	Funzione <code>recFree</code>	8
4.2.4	Funzione <code>parse</code>	9
4.2.5	Funzione <code>treeToStr</code>	10
4.2.6	Funzione <code>simplify</code>	11
4.2.7	Funzioni <code>gcd</code> e <code>lcm</code>	12
4.2.8	Funzione <code>getLCM</code>	13
4.2.9	Funzione <code>normalize</code>	14
4.2.10	Funzione <code>minInf</code>	16
4.2.11	Funzione <code>calcm</code>	17
4.2.12	Funzione <code>boundaryPoints</code>	17
4.2.13	Funzione <code>newFormula</code>	19
4.2.14	Funzione <code>eval</code>	20
5	Utilizzo	21
5.1	Il programma <code>test.c</code>	21
5.2	Il Makefile	21
5.3	Valutazione e soddisfacibilità	22

Riferimenti bibliografici

- 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>.
- 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.
- 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>.