

Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

1 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 come ingresso una formula del tipo $\exists x. \varphi$ e come 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 é 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.

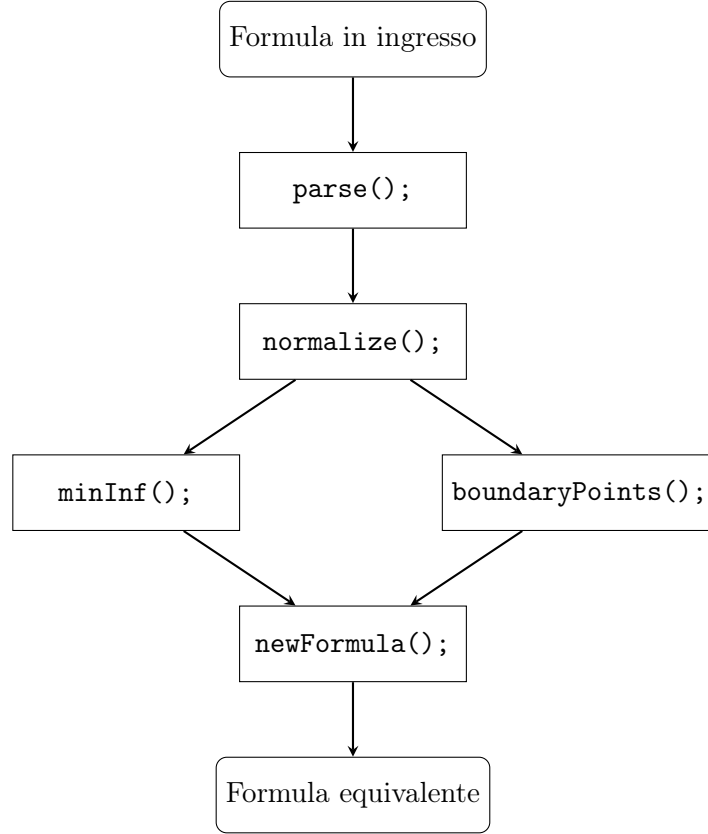
3 Implementazione

Il software è stato scritto nel linguaggio C rispettando lo standard C99,⁴ in questo capitolo verrà effettuata una discussione riguardo l'implementazione.

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>.

⁴ISO. *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>.

3.1 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-LIB⁵ 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)`.

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.

⁵Clark Barrett, Pascal Fontaine e Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

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.

3.2 Analisi del codice

Quella che viene presentata qui è un'analisi dettagliata del codice sorgente del programma riga per riga.

```

521 char* cooper(char* wff, char* var) {
522     t_syntaxTree* tree, *minf, *f;
523     char* str;
524
525     tree = parse(wff); //Genera l'albero sintattico a partire dalla stringa
526     normalize(tree, var); //Trasforma l'albero di tree
527     minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
528     f = newFormula(tree, minf, var); //Restituisce la formula equivalente
529     simplify(f); //opzionale
530     str = treeToStr(f); //Genera la stringa a partire dall'albero
531
532     recFree(tree); //Libera la memoria
533     recFree(minf);
534     recFree(f);
535
536     return str;
537 }
```

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`.

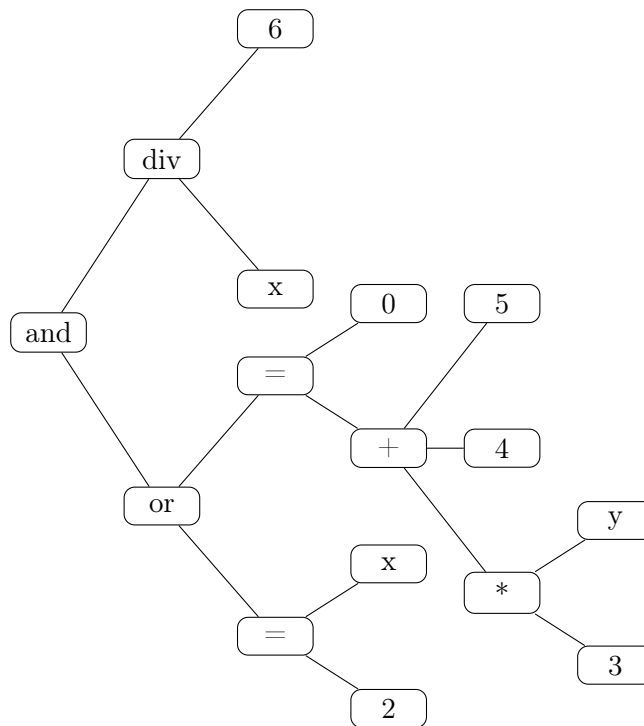
```

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

```

204 void recFree(t_syntaxTree* tree) {
205     for (int i=0; i<tree->nodesLen; i++) {
206         recFree(tree->nodes[i]);
207     }
208
209     free(tree->nodes);
210     free(tree);
211 }

```

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.

```

23 t_syntaxTree* buildTree(int first, char** tokens) {
24     t_syntaxTree* tree = malloc(sizeof(t_syntaxTree));
25     tree->nodes = NULL;
26     int open;
27
28     if (tokens[first][0] == '(') {
29         first++;
30         tree->nodesLen = 0;
31         strcpy(tree->nodeName, tokens[first]);
32         open = 1;

```

```

33
34     do {
35         first++;
36
37         if (open == 1 && tokens[first][0]!='(') {
38             tree->nodesLen++;
39             tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
40             tree->nodes[tree->nodesLen-1] = buildTree(first, tokens);
41         }
42
43         if (tokens[first][0] == '(') open++;
44
45         if (tokens[first][0] == ')') open--;
46     } while (open != 0);
47 }
48
49 else {
50     strcpy(tree->nodeName, tokens[first]);
51     tree->nodesLen = 0;
52     tree->nodes = NULL;
53 }
54
55 return tree;
56 }
57
58
59 t_syntaxTree* parse(char* wff) {
60     char* wffSpaced = malloc(sizeof(char));
61     wffSpaced[0] = wff[0];
62     int j = 1;
63
64     for (int i = 1; i < strlen(wff) + 1; i++) {
65
66         if (wff[i - 1] == '(') {
67             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
68             wffSpaced[j] = ' ';
69             wffSpaced[j+1] = wff[i];
70             j += 2;
71         }
72
73         else if (wff[i + 1] == ')') {
74             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 2));
75             wffSpaced[j] = wff[i];
76             wffSpaced[j + 1] = ' ';
77             j += 2;
78         }
79
80         else {
81             wffSpaced = realloc(wffSpaced, sizeof(char) * (j + 1));
82             wffSpaced[j] = wff[i];

```

```

83     j++;
84 }
85 }
86
87 char* token;
88 int nTokens = 1;
89 char** tokens = malloc(sizeof(char *));
90 tokens[0] = strtok(wffSpaced, " ");
91
92 while ((token = strtok(NULL, " ")) != NULL) {
93     nTokens++;
94     tokens = realloc(tokens, sizeof(char *) * nTokens);
95     tokens[nTokens - 1] = token;
96 }
97
98 t_syntaxTree* syntaxTree = buildTree(0, tokens);
99
100 free(wffSpaced);
101 free(tokens);
102
103 return syntaxTree;
104 }

```

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.

```

483 int recTreeToStr(t_syntaxTree* t, char** str, int len) {
484     if (t->nodesLen == 0) {
485         int nLen = len + strlen(t->nodeName);
486         *str = realloc(*str, sizeof(char) * nLen);
487         strcat(*str, t->nodeName);
488         return nLen;
489     }
490
491     else {
492         int nLen = len + strlen(t->nodeName) + 1;
493         *str = realloc(*str, sizeof(char) * nLen);
494         strcat(*str, "(");
495         strcat(*str, t->nodeName);
496
497         for (int i=0; i<t->nodesLen; i++) {
498             nLen++;
499             *str = realloc(*str, sizeof(char) * nLen);
500             strcat(*str, " ");
501             nLen = recTreeToStr(t->nodes[i], str, nLen);
502         }
503

```



```

504     nLen++;
505     *str = realloc(*str, sizeof(char) * nLen);
506     strcat(*str, ")");
507
508     return nLen;
509 }
510 }
511
512
513 char* treeToStr(t_syntaxTree* tree) {
514     char* str=malloc(sizeof(char));
515     str[0] = '\0';
516     recTreeToStr(tree, &str, 1);
517     return str;
518 }

```

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.

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     tree = parse(wff); //Genera l'albero sintattico a partire dalla stringa
526     normalize(tree, var); //Trasforma l'albero di tree
527     minf = minInf(tree, var); //Restituisce l'albero di  $\varphi_{-\infty}$ 
528     f = newFormula(tree, minf, var); //Restituisce la formula equivalente
529     simplify(f); //opzionale
530     str = treeToStr(f); //Genera la stringa a partire dall'albero

```

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.

```

442 void simplify(t_syntaxTree* t) {
443     if (t->nodesLen != 0) {
444         int simplified = 0;
445
446         if (strcmp(t->nodeName, "and") == 0) {
447             for(int i=0; i<t->nodesLen; i++) {
448                 if (strcmp(t->nodes[i]->nodeName, "false") == 0) {
449                     simplified = 1;
450
451                     for (int j=0; j<t->nodesLen; j++)
452                         recFree(t->nodes[j]);
453
454                     strcpy(t->nodeName, "false");
455                     t->nodesLen = 0;
456                     break;
457                 }
458             }
459         }
460     }

```

```

461     if (strcmp(t->nodeName, "or") == 0) {
462         for(int i=0; i<t->nodesLen; i++) {
463             if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
464                 simplified = 1;
465
466                 for (int j=0; j<t->nodesLen; j++)
467                     recFree(t->nodes[j]);
468
469                 strcpy(t->nodeName, "true");
470                 t->nodesLen = 0;
471                 break;
472             }
473         }
474     }
475
476     if (!simplified)
477         for(int i=0; i<t->nodesLen; i++)
478             simplify(t->nodes[i]);
479 }
480 }

```

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`.

```

6  long int gcd(long int a, long int b) {
7      return b == 0 ? a : gcd(b, a % b);
8  }
9
10
11 long int lcm(long int a, long int b) {
12     return abs((a / gcd(a, b)) * b);
13 }

```

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 Euclide⁶ mentre il secondo è dato banalmente dalla seguente.

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

⁶Euclid. *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.

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.

```

107 int getLCM(t_syntaxTree* tree, char* var) {
108     if (tree->nodeName[0] == '*') {
109         if (strcmp(((t_syntaxTree *)tree->nodes[1])->nodeName, var) == 0) {
110             return atoi(((t_syntaxTree *) tree->nodes[0])->nodeName);
111         }
112     }
113
114     int l = 1;
115
116     for(int i=0; i<tree->nodesLen; i++) {
117         l = lcm(l, getLCM((t_syntaxTree *) tree->nodes[i], var));
118     }
119
120     return l;
121 }

```

`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,⁷ 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:

⁷Ghilardi, *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) agginuge anche un opportuno predicato di divisibilità come specificato nell'algoritmo.

```
124 void normalize(t_syntaxTree* tree, char* var) {
125     int lcm = getLCM(tree, var);
126     int c;
127
128     for (int i=0; i<tree->nodesLen; i++) {
129         if (strcmp("=", tree->nodes[i]->nodeName) == 0 ||
130             strcmp("div", tree->nodes[i]->nodeName) == 0) {
131             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
132
133             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
134                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
135                     c = atoi(addends[j]->nodes[0]->nodeName);
136             }
137
138             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
139                 if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
140                     strcpy(addends[j]->nodeName, var);
141                     free(addends[j]->nodes[0]);
142                     free(addends[j]->nodes[1]);
143                     addends[j]->nodesLen = 0;
144                 }
145                 else {
146                     sprintf(addends[j]->nodes[0]->nodeName,
147                             "%d",
148                             atoi(addends[j]->nodes[0]->nodeName)*lcm/c);
149                 }
150             }
151
152             sprintf(tree->nodes[i]->nodes[1]->nodeName,
153                     "%d",
154                     atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/c);
155         }
156
157         else if (strcmp(">", tree->nodes[i]->nodeName) == 0) {
```

```

158     t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
159
160     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
161         if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
162             c = atoi(addends[j]->nodes[0]->nodeName);
163     }
164
165     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
166         if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
167             if (c > 0) strcpy(addends[j]->nodeName, "");
168             else strcpy(addends[j]->nodeName, "-");
169             strcat(addends[j]->nodeName, var);
170             free(addends[j]->nodes[0]);
171             free(addends[j]->nodes[1]);
172             addends[j]->nodesLen = 0;
173         }
174         else {
175             sprintf(addends[j]->nodes[0]->nodeName,
176                     "%d",
177                     atoi(addends[j]->nodes[0]->nodeName)*1cm/abs(c));
178         }
179     }
180
181     sprintf(tree->nodes[i]->nodes[1]->nodeName,
182             "%d",
183             atoi(tree->nodes[i]->nodes[1]->nodeName)*1cm/abs(c));
184 }
185 }
186
187 tree->nodesLen++;
188 tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
189 tree->nodes[tree->nodesLen-1] = malloc(sizeof(t_syntaxTree));
190 strcpy(tree->nodes[tree->nodesLen-1]->nodeName, "div");
191 tree->nodes[tree->nodesLen-1]->nodesLen = 2;
192 tree->nodes[tree->nodesLen-1]->nodes = malloc(sizeof(t_syntaxTree*) * 2);
193 tree->nodes[tree->nodesLen-1]->nodes[0] = malloc(sizeof(t_syntaxTree));
194 tree->nodes[tree->nodesLen-1]->nodes[1] = malloc(sizeof(t_syntaxTree));
195 tree->nodes[tree->nodesLen-1]->nodes[0]->nodesLen = 0;
196 tree->nodes[tree->nodesLen-1]->nodes[0]->nodes = NULL;
197 tree->nodes[tree->nodesLen-1]->nodes[1]->nodesLen = 0;
198 tree->nodes[tree->nodesLen-1]->nodes[1]->nodes = NULL;
199 strcpy(tree->nodes[tree->nodesLen-1]->nodes[0]->nodeName, var);
200 sprintf(tree->nodes[tree->nodesLen-1]->nodes[1]->nodeName, "%d", 1cm);
201 }

```

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.

```

233 t_syntaxTree* minInf(t_syntaxTree* tree, char* var) {
234     t_syntaxTree* nTree = recCopy(tree);
235

```

```

236 char minvar[16];
237 minvar[0] = '\0';
238 strcpy(minvar, "-");
239 strcat(minvar, var);
240
241 for (int i=0; i<nTree->nodesLen; i++) {
242     if (strcmp(">", nTree->nodes[i]->nodeName) == 0) {
243         t_syntaxTree** addends = nTree->nodes[i]->nodes[0]->nodes;
244
245         for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) {
246             if (strcmp(addends[j]->nodeName, var) == 0)
247                 strcpy(nTree->nodes[i]->nodeName, "false");
248             else if (strcmp(addends[j]->nodeName, minvar) == 0)
249                 strcpy(nTree->nodes[i]->nodeName, "true");
250         }
251
252         for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
253             recFree(nTree->nodes[i]->nodes[j]);
254
255         free(nTree->nodes[i]->nodes);
256         nTree->nodes[i]->nodesLen = 0;
257         nTree->nodes[i]->nodes = NULL;
258     }
259
260     else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) {
261         for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
262             recFree(nTree->nodes[i]->nodes[j]);
263
264         free(nTree->nodes[i]->nodes);
265         nTree->nodes[i]->nodesLen = 0;
266         nTree->nodes[i]->nodes = NULL;
267         strcpy(nTree->nodes[i]->nodeName, "false");
268     }
269 }
270
271 return nTree;
272 }

```

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.

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

```

287 int calcm(t_syntaxTree* tree, char* var) {
288     int m=1;
289

```

```

290     for(int i=0; i<tree->nodesLen; i++) {
291         if(strcmp(tree->nodes[i]->nodeName, "div") == 0) {
292
293             if(strcmp(tree->nodes[i]->nodes[0]->nodeName, var) == 0)
294                 m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
295
296             else if(strcmp(tree->nodes[i]->nodes[0]->nodeName, "+") == 0) {
297                 for(int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
298                     if (strcmp(tree->nodes[i]->nodes[0]->nodes[j]->nodeName, var) == 0) {
299                         m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
300                         break;
301                     }
302                 }
303             }
304         }
305     }
306
307     return m;
308 }

```

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.

```

311 t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var) {
312     char str[16];
313     str[0] = '\0';
314     t_syntaxTree* bPoints = malloc(sizeof(t_syntaxTree));
315     bPoints->nodes = NULL;
316     strcpy(bPoints->nodeName, "bPoints"); //solo per debugging
317     bPoints->nodesLen = 0;
318
319     for(int i=0; i<tree->nodesLen; i++) {
320         if (strcmp(tree->nodes[i]->nodeName, "=") == 0) {
321             t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
322
323             for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
324                 if (strcmp(var, addends[j]->nodeName) == 0) {
325                     bPoints->nodesLen++;
326                     bPoints->nodes = realloc(bPoints->nodes, sizeof(t_syntaxTree *) * bPoints->nodesLen);
327                     t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
328                     bp->nodes = NULL;
329                     strcpy(bp->nodeName, "+");
330                     bp->nodesLen = 0;
331
332                     for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
333                         if (strcmp(var, addends[k]->nodeName) != 0) {
334                             bp->nodesLen++;
335                             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);

```

```

336         bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
337         sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
338         strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
339     }
340 }
341
342 bp->nodesLen++;
343 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
344 bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
345 bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
346 bp->nodes[bp->nodesLen - 1]->nodes = NULL;
347 sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
348 strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);
349
350 bPoints->nodes[bPoints->nodesLen-1] = bp;
351 break;
352 }
353 }
354 }
355
356 if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
357     t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
358
359     for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
360         if (strcmp(var, addends[j]->nodeName) == 0) {
361             bPoints->nodesLen++;
362             bPoints->nodes = realloc(bPoints->nodes, sizeof(t_syntaxTree *) * bPoints->nodesLen);
363             t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
364             bp->nodes = NULL;
365             strcpy(bp->nodeName, "+");
366             bp->nodesLen = 0;
367
368             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
369                 if (strcmp(var, addends[k]->nodeName) != 0) {
370                     bp->nodesLen++;
371                     bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
372                     bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
373                     sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
374                     strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
375                 }
376             }
377
378             bp->nodesLen++;
379             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
380             bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
381             bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
382             bp->nodes[bp->nodesLen - 1]->nodes = NULL;
383             sprintf(str, "%d", +atoi(tree->nodes[i]->nodes[1]->nodeName));
384             strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);
385

```



```

386         bPoints->nodes[bPoints->nodesLen-1] = bp;
387         break;
388     }
389 }
390 }
391 }
392
393 return bPoints;
394 }

```

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`.

```

397 t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
398     int m = calcm(minf, var);
399     t_syntaxTree* val;
400     char str[16];
401     t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree));
402     strcpy(nTree->nodeName, "or");
403     nTree->nodesLen = 0;
404     nTree->nodes = NULL;
405
406     t_syntaxTree* t;
407     t_syntaxTree* bp;
408     t_syntaxTree *bPts = boundaryPoints(tree, var);
409
410     for(int i=1; i<=m; i++) {
411         nTree->nodesLen++;
412         nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
413         t = recCopy(minf);
414         val = malloc(sizeof(t_syntaxTree));
415         sprintf(str, "%d", i);
416         strcpy(val->nodeName, str);
417         val->nodesLen = 0;
418         val->nodes = NULL;
419         eval(t, var, val);
420         recFree(val);
421         nTree->nodes[nTree->nodesLen-1] = t;
422
423         for(int j=0; j<bPts->nodesLen; j++) {
424             nTree->nodesLen++;
425             nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
426             t = recCopy(tree);
427             bp = recCopy(bPts->nodes[j]);
428             sprintf(str, "%d", i+atoi(bp->nodes[bp->nodesLen-1]->nodeName));
429             strcpy(bp->nodes[bp->nodesLen-1]->nodeName, str);
430             eval(t, var, bp);
431             recFree(bp);
432

```

```

433     nTree->nodes[nTree->nodesLen-1] = t;
434 }
435 }
436
437 recFree(bPts);
438 return nTree;
439 }

```

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.

```

275 void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val) {
276     for (int i=0; i<tree->nodesLen; i++) {
277         if (strcmp(tree->nodes[i]->nodeName, var) == 0) {
278             recFree(tree->nodes[i]);
279             tree->nodes[i] = recCopy(val);
280         }
281         else
282             eval(tree->nodes[i], var, val);
283     }
284 }

```

4 Utilizzo

In questo capitolo verranno forniti alcuni semplici esempi di utilizzo.

4.1 Script ausiliari

4.2 Esempi pratici

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	Implementazione	3
3.1	Struttura e design	4
3.2	Analisi del codice	5
4	Utilizzo	18
4.1	Script ausiliari	18
4.2	Esempi pratici	18

Riferimenti bibliografici

- Barrett, Clark, Pascal Fontaine e Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.
- 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.
- Ghilardi, Silvio. *MCMT: Model Checker Modulo Theories*. <http://users.mat.unimi.it/users/ghilardi/mcmt/>. 2018.
- 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>.
- 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>.