Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

2 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 divisibiltà 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 $\mathcal{A}_{\mathbb{Z}}^\star$ ottenuta da $\mathcal{A}_{\mathbb{Z}}$.

Nel 1930 Mojżesz Presburger ha esibito un algoritmo di eliminazione dei quantificatori[2] per $T_{\mathbb{Z}}^*$ e nel 1972 Cooper ha fornito una versione migliorata basata sull'eliminazione dei quantificatori da formule nella forma $\exists\, x \,.\, \varphi$, dove φ è una formula senza quantificatori arbitraria.

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 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 generalitá 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 $s < t + 1$).

[1] Silvio Ghilardi. *MCMT: Model Checker Modulo Theories*. `http://users.mat.unimi.it/users/ghilardi/mcmt/`. 2018.

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

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

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 divisibiltà 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 divisbilità $D_k(t)$ 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.

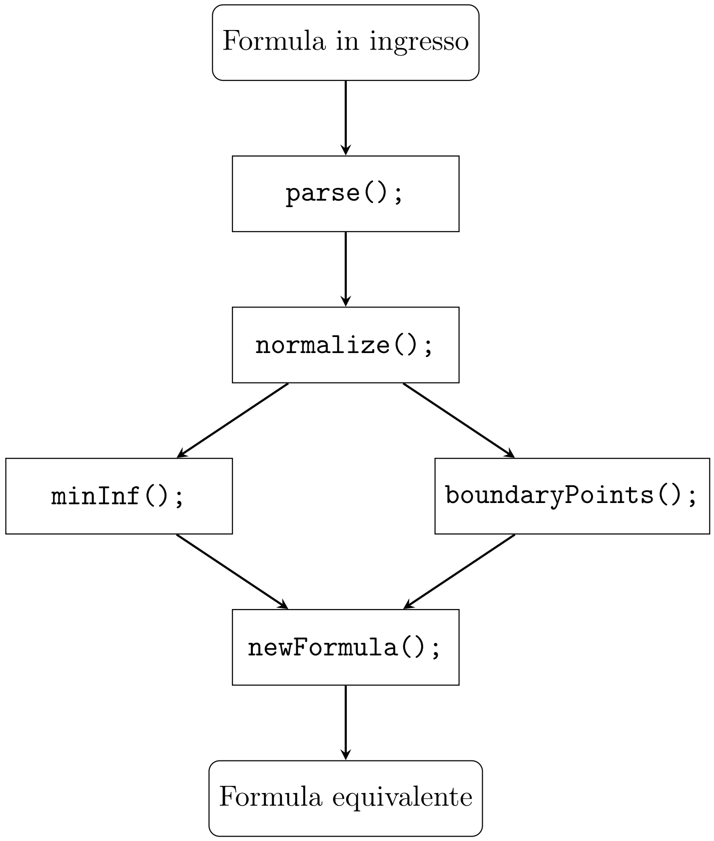
3 Implementazione

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

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

[4] 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`.

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-LIB[5] 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.

[5]Clark 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.

```
char* cooper(char* wff, char* var) {
  t_syntaxTree* tree, *minf, *f;
  char* str;

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

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

  return str;
}
```

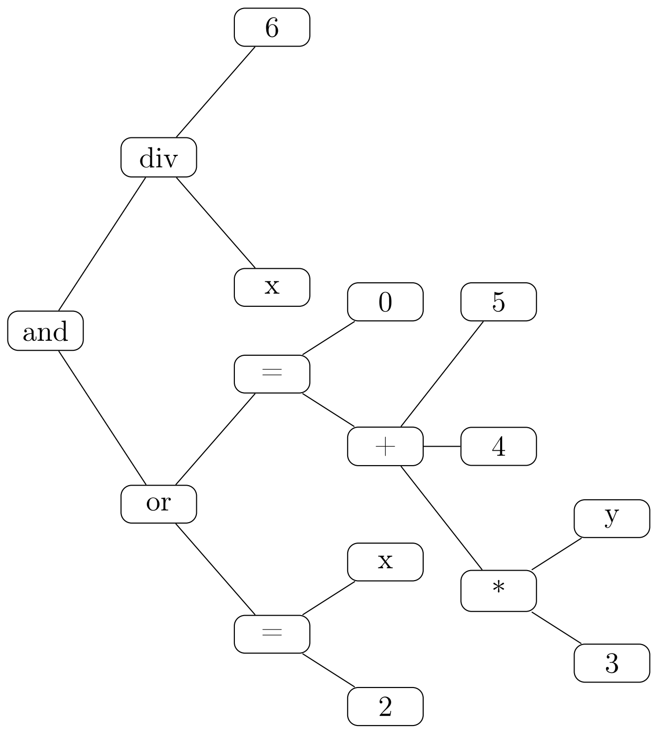
Alla luce di quanto detto precedentemente il funzionamento di `cooper()` risulta autoesplicativo. É quindi arrivato il momento di esporre la segnatura completa del tipo composto `t_syntaxTree`.

```
typedef struct t_syntaxTree {
  char nodeName[16];
  int nodesLen;
  struct t_syntaxTree** nodes;
} t_syntaxTree;
```

Trattasi di un record definito ricorsivivamente avente 3 campi:

- `char nodeName[16]` è una stringa di lunghezza fissata posta arbritrariamente a 16 caratteri, è il nome del nodo nell'albero sintattico.
- `int nodesLen` è il numero di figli del nodo in questione
- `t_syntaxTree** nodes` è un array di puntatori ad altri nodi

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



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

```
void recFree(t_syntaxTree* tree) {
  for (int i=0; i<tree->nodesLen; i++) {
    recFree(tree->nodes[i]);
  }

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

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

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

```
t_syntaxTree* buildTree(int first, char** tokens) {
  t_syntaxTree* tree = malloc(sizeof(t_syntaxTree));
  tree->nodes = NULL;
  int open;

  if (tokens[first][0] == '(') {
    first++;
    tree->nodesLen = 0;
    strcpy(tree->nodeName, tokens[first]);
    open = 1;
```

```
    do {
      first++;

      if (open == 1 && tokens[first][0]!=')') {
        tree->nodesLen++;
        tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
        tree->nodes[tree->nodesLen-1] = buildTree(first, tokens);
      }

      if (tokens[first][0] == '(') open++;

      if (tokens[first][0] == ')') open--;
    } while (open != 0);
  }

  else {
    strcpy(tree->nodeName, tokens[first]);
    tree->nodesLen = 0;
    tree->nodes = NULL;
  }

  return tree;
}


t_syntaxTree* parse(char* wff) {
  char* wffSpaced = malloc(sizeof(char));
  wffSpaced[0] = wff[0];
  int j = 1;

  for (int i = 1; i < strlen(wff) + 1; i++) {

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

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

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

```
      j++;
    }
  }

  char* token;
  int nTokens = 1;
  char** tokens = malloc(sizeof(char *));
  tokens[0] = strtok(wffSpaced, " ");

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

  t_syntaxTree* syntaxTree = buildTree(0, tokens);

  free(wffSpaced);
  free(tokens);

  return syntaxTree;
}
```

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

```
int recTreeToStr(t_syntaxTree* t, char** str, int len) {
  if (t->nodesLen == 0) {
    int nLen = len + strlen(t->nodeName);
    *str = realloc(*str, sizeof(char) * nLen);
    strcat(*str, t->nodeName);
    return nLen;
  }

  else {
    int nLen = len + strlen(t->nodeName) + 1;
    *str = realloc(*str, sizeof(char) * nLen);
    strcat(*str, "(");
    strcat(*str, t->nodeName);

    for (int i=0; i<t->nodesLen; i++) {
      nLen++;
      *str = realloc(*str, sizeof(char) * nLen);
      strcat(*str, " ");
      nLen = recTreeToStr(t->nodes[i], str, nLen);
    }

```

```c
    nLen++;
    *str = realloc(*str, sizeof(char) * nLen);
    strcat(*str, ")");

    return nLen;
  }
}


char* treeToStr(t_syntaxTree* tree) {
  char* str=malloc(sizeof(char));
  str[0] = '\0';
  recTreeToStr(tree, &str, 1);
  return str;
}
```

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

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

```c
  tree = parse(wff); //Genera l'albero sintattico a partire dalla stringa
  normalize(tree, var); //Trasforma l'albero di tree
  minf = minInf(tree, var); //Restituisce l'albero di $\varphi_{-\infty}$
  f = newFormula(tree, minf, var); //Restituisce la formula equivalente
  simplify(f); //opzionale
  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.

```c
void simplify(t_syntaxTree* t) {
  if (t->nodesLen != 0) {
    int simplified = 0;

    if (strcmp(t->nodeName, "and") == 0) {
      for(int i=0; i<t->nodesLen; i++) {
        if (strcmp(t->nodes[i]->nodeName, "false") == 0) {
          simplified = 1;

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

          strcpy(t->nodeName, "false");
          t->nodesLen = 0;
          break;
        }
      }
    }

```

```
    if (strcmp(t->nodeName, "or") == 0) {
      for(int i=0; i<t->nodesLen; i++) {
        if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
          simplified = 1;

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

          strcpy(t->nodeName, "true");
          t->nodesLen = 0;
          break;
        }
      }
    }

    if (!simplified)
      for(int i=0; i<t->nodesLen; i++)
        simplify(t->nodes[i]);
  }
}
```

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`.

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


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

Come è facile immaginare tali funzioni effettuano semplicemente il calcolo del massimo comun divisore e del minimo comune multiplo. Il primo viene svolto efficacemente dall'algoritmo di Euclide[6] mentre il secondo è dato banalmente dalla seguente.

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

[6] 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.

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.

```
int getLCM(t_syntaxTree* tree, char* var) {
  if (tree->nodeName[0] == '*') {
    if (strcmp(((t_syntaxTree *)tree->nodes[1])->nodeName, var) == 0) {
      return atoi(((t_syntaxTree *) tree->nodes[0])->nodeName);
    }
  }

  int l = 1;

  for(int i=0; i<tree->nodesLen; i++) {
    l = lcm(l, getLCM((t_syntaxTree *) tree->nodes[i], var));
  }

  return l;
}
```

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

Risulta quindi ora utile discutere quale sia la forma esatta dell'input gestito dal programma, molte assunzioni che 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:

[7] Ghilardi, *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.

```c
void normalize(t_syntaxTree* tree, char* var) {
  int lcm = getLCM(tree, var);
  int c;

  for (int i=0; i<tree->nodesLen; i++) {
    if (strcmp("=", tree->nodes[i]->nodeName) == 0 ||
        strcmp("div", tree->nodes[i]->nodeName) == 0) {
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
          c = atoi(addends[j]->nodes[0]->nodeName);
      }

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
          strcpy(addends[j]->nodeName, var);
          free(addends[j]->nodes[0]);
          free(addends[j]->nodes[1]);
          addends[j]->nodesLen = 0;
        }
        else {
          sprintf(addends[j]->nodes[0]->nodeName,
                  "%d",
                  atoi(addends[j]->nodes[0]->nodeName)*lcm/c);
        }
      }

      sprintf(tree->nodes[i]->nodes[1]->nodeName,
              "%d",
              atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/c);
    }

    else if (strcmp(">", tree->nodes[i]->nodeName) == 0) {
```

```c
      t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
          c = atoi(addends[j]->nodes[0]->nodeName);
      }

      for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
          if(c > 0) strcpy(addends[j]->nodeName, "");
          else strcpy(addends[j]->nodeName, "-");
          strcat(addends[j]->nodeName, var);
          free(addends[j]->nodes[0]);
          free(addends[j]->nodes[1]);
          addends[j]->nodesLen = 0;
        }
        else {
          sprintf(addends[j]->nodes[0]->nodeName,
                  "%d",
                  atoi(addends[j]->nodes[0]->nodeName)*lcm/abs(c));
        }
      }

      sprintf(tree->nodes[i]->nodes[1]->nodeName,
              "%d",
              atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/abs(c));
    }
  }

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

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

```c
t_syntaxTree* minInf(t_syntaxTree* tree, char* var) {
  t_syntaxTree* nTree = recCopy(tree);

```

```c
  char minvar[16];
  minvar[0] = '\0';
  strcpy(minvar, "-");
  strcat(minvar, var);

  for (int i=0; i<nTree->nodesLen; i++) {
    if (strcmp(">", nTree->nodes[i]->nodeName) == 0) {
      t_syntaxTree** addends = nTree->nodes[i]->nodes[0]->nodes;

      for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) {
        if (strcmp(addends[j]->nodeName, var) == 0)
          strcpy(nTree->nodes[i]->nodeName, "false");
        else if (strcmp(addends[j]->nodeName, minvar) == 0)
          strcpy(nTree->nodes[i]->nodeName, "true");
      }

      for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
        recFree(nTree->nodes[i]->nodes[j]);

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

    else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) {
      for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
        recFree(nTree->nodes[i]->nodes[j]);

      free(nTree->nodes[i]->nodes);
      nTree->nodes[i]->nodesLen = 0;
      nTree->nodes[i]->nodes = NULL;
      strcpy(nTree->nodes[i]->nodeName, "false");
    }
  }

  return nTree;
}
```

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

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

```c
int calcm(t_syntaxTree* tree, char* var) {
  int m=1;

```

```c
  for(int i=0; i<tree->nodesLen; i++) {
    if(strcmp(tree->nodes[i]->nodeName, "div") == 0) {

      if(strcmp(tree->nodes[i]->nodes[0]->nodeName, var) == 0)
        m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));

      else if(strcmp(tree->nodes[i]->nodes[0]->nodeName, "+") == 0) {
        for(int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
          if (strcmp(tree->nodes[i]->nodes[0]->nodes[j]->nodeName, var) == 0) {
            m = lcm(m, atoi(tree->nodes[i]->nodes[1]->nodeName));
            break;
          }
        }
      }
    }
  }

  return m;
}
```

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

```c
t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var) {
  char str[16];
  str[0] = '\0';
  t_syntaxTree* bPoints = malloc(sizeof(t_syntaxTree));
  bPoints->nodes = NULL;
  strcpy(bPoints->nodeName, "bPoints"); //solo per debugging
  bPoints->nodesLen = 0;

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

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

          for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
            if (strcmp(var, addends[k]->nodeName) != 0) {
              bp->nodesLen++;
              bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
```

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

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

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

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

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

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

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

```

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

  return bPoints;
}
```

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

```c
t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
  int m = calcm(minf, var);
  t_syntaxTree* val;
  char str[16];
  t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree));
  strcpy(nTree->nodeName, "or");
  nTree->nodesLen = 0;
  nTree->nodes = NULL;

  t_syntaxTree* t;
  t_syntaxTree* bp;
  t_syntaxTree *bPts = boundaryPoints(tree, var);

  for(int i=1; i<=m; i++) {
    nTree->nodesLen++;
    nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
    t = recCopy(minf);
    val = malloc(sizeof(t_syntaxTree));
    sprintf(str, "%d", i);
    strcpy(val->nodeName, str);
    val->nodesLen = 0;
    val->nodes = NULL;
    eval(t, var, val);
    recFree(val);
    nTree->nodes[nTree->nodesLen-1] = t;

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

```

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

  recFree(bPts);
  return nTree;
}
```

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

Trattasi ovviamente della funzione `void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val)`, si osservi anche qui come 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.

```
void eval(t_syntaxTree* tree, char* var, t_syntaxTree* val) {
  for (int i=0; i<tree->nodesLen; i++) {
    if (strcmp(tree->nodes[i]->nodeName, var) == 0) {
      recFree(tree->nodes[i]);
      tree->nodes[i] = recCopy(val);
    }
    else
      eval(tree->nodes[i], var, val);
  }
}
```

4 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.

Si consideri il seguente programma di esempio contenuto in `test.c`:

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

int main(int argc, char** argv) {
    if (argc == 3)
        printf("%s", cooper(argv[1], argv[2]));
    else
        printf("Numero errato di argomenti!");

    return 0;
}
```

Si consideri ora il seguente `makefile`:

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

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

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

run: test #esegue test e restituisce il tempo impiegato
	time ./test $(wff) $(var)

sat: test sat.py #verifica la soddisfacibilità della formula generata grazie a yices
	./sat.py $(wff) $(vars)

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

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

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

clean:
	rm -f *.o
	rm test
```

4.1 Script ausiliari

4.2 Esempi pratici

Indice

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: `10.1080/01445340910883718 7`. URL: `https://doi-org.pros.lib.unimi.it:2050/10.1080/014453409108837187`.