Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

18 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 $\mathcal{A}^*_{\mathbb{Z}}$ 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 in ingresso una formula del tipo $\exists\, x\,.\,\varphi$ e in 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 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 $s < t + 1$).
- Le diseguaglianze negate della forma $\neg(s < t)$ vengono sostituite con $t < s + 1$.

[1] **mcmt**.
[2] **presburger**.

- 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 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 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.

2.6 Complessità computazionale

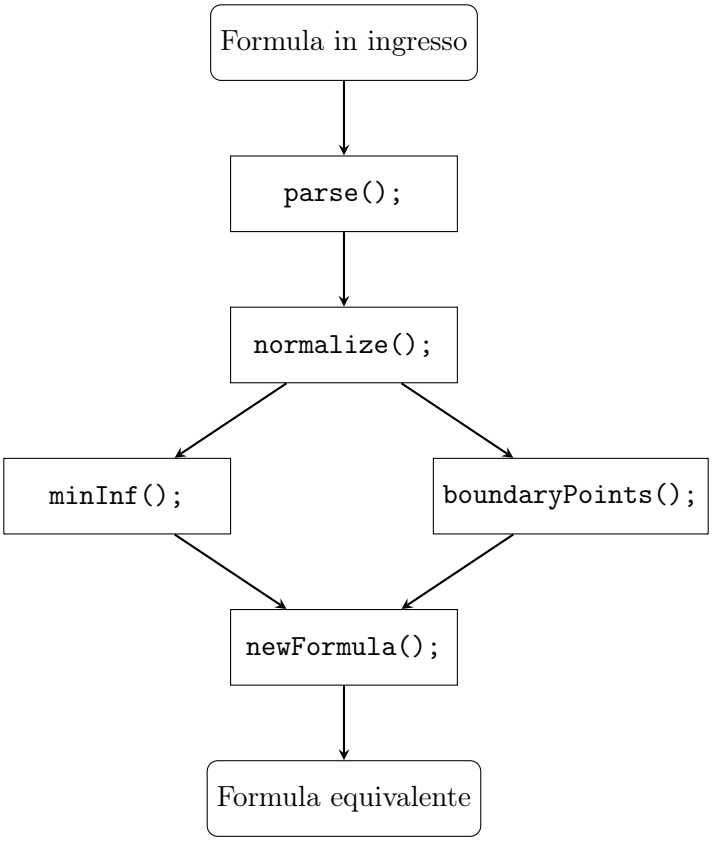
Si accenni solamente al notevole risultato dovuto a Fischer e Rabin,[4] nel 1974 mostrarono infatti che data n la lunghezza di una formula nell'aritmetica di Presburger, ogni problema decisionale ha complessità temporale $2^{2^{cn}}$ nel caso peggiore, per qualche costante $c \geq 0$. Ovvero l'algoritmo di Cooper è obbligatoriamente NP-difficile avendo una complessità almeno esponenziale doppia.

[3] **cooper**.
[4] **complexity**.

3 Implementazione

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

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[6] 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)`.

[5] **c99**.
[6] **smtlib**.

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.

3.2 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.

3.2.1 Funzione `cooperToStr`

```
585 char* cooperToStr(char* wff, char* var) {
586   t_syntaxTree* tree, *minf, *f;
587   char* str;
588 
589   tree = parse(wff); //Genera l'albero sintattico a partire dalla stringa
590   normalize(tree, var); //Trasforma l'albero di tree
591   minf = minInf(tree, var); //Restituisce l'albero di φ_-∞
592   f = newFormula(tree, minf, var); //Restituisce la formula equivalente
593   //simplify(f); //opzionale
594   str = treeToStr(f); //Genera la stringa a partire dall'albero
595 
596   recFree(tree); //Libera la memoria
597   recFree(minf);
598   recFree(f);
599 
600   return str;
601 }
```

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

3.2.2 Segnatura di `t_syntaxTree`

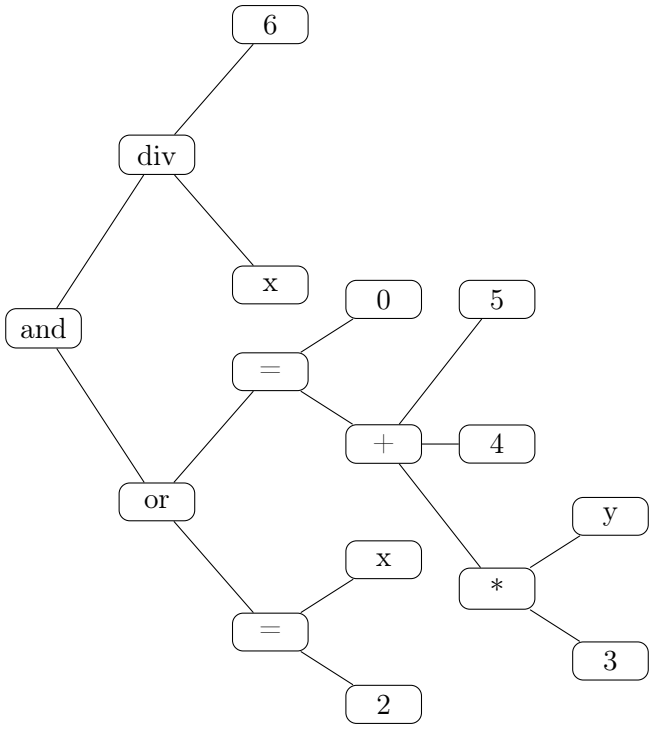
```
17 typedef struct t_syntaxTree {
18   char nodeName[16];
19   int nodesLen;
20   struct t_syntaxTree** nodes;
21 } t_syntaxTree;
```

Trattasi di un record definito ricorsivivamente 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 segente diagramma.



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

3.2.3 Funzione `recFree`

```
268 void recFree(t_syntaxTree* tree) {
269   for (int i=0; i<tree->nodesLen; i++) {
270     recFree(tree->nodes[i]);
271   }
272
273   free(tree->nodes);
274   free(tree);
275 }
```

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

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

3.2.4 Funzione parse

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

```
155   }
156
157   if (countPar != 0)
158     ERROR("Parsing error: the number of parentheses is not even");
159
160   t_syntaxTree* syntaxTree = buildTree(0, tokens);
161
162   checkTree(syntaxTree); //chiama exit() se l'albero non va bene
163
164   free(wffSpaced);
165   free(tokens);
166
167   return syntaxTree;
168 }
```

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

3.2.5 Funzione `treeToStr`

```
577 char* treeToStr(t_syntaxTree* tree) {
578   char* str=malloc(sizeof(char));
579   str[0] = '\0';
580   recTreeToStr(tree, &str, 1);
581   return str;
582 }
```

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.

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

```
563         *str = realloc(*str, sizeof(char) * nLen);
564         strcat(*str, " ");
565         nLen = recTreeToStr(t->nodes[i], str, nLen);
566       }
567
568       nLen++; //nLen++;
569       *str = realloc(*str, sizeof(char) * nLen);
570       strcat(*str, ")");
571
572       return nLen;
573     }
574   }
```

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       if (strcmp(t->nodeName, "or") == 0) {
526         for(int i=0; i<t->nodesLen; i++) {
527           if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
528             simplified = 1;
529
530             for (int j=0; j<t->nodesLen; j++)
```

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

3.2.6 Funzione `simplify`

```
506 void simplify(t_syntaxTree* t) {
507   if (t->nodesLen != 0) {
508     int simplified = 0;
509
510     if (strcmp(t->nodeName, "and") == 0) {
511       for(int i=0; i<t->nodesLen; i++) {
512         if (strcmp(t->nodes[i]->nodeName, "false") == 0) {
513           simplified = 1;
514
515           for (int j=0; j<t->nodesLen; j++)
516             recFree(t->nodes[j]);
517
518           strcpy(t->nodeName, "false");
519           t->nodesLen = 0;
520           break;
521         }
522       }
523     }
524
525     if (strcmp(t->nodeName, "or") == 0) {
526       for(int i=0; i<t->nodesLen; i++) {
527         if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
528           simplified = 1;
```

```
529
530             for (int j=0; j<t->nodesLen; j++)
531               recFree(t->nodes[j]);
532
533             strcpy(t->nodeName, "true");
534             t->nodesLen = 0;
535             break;
536           }
537         }
538       }
539
540       if (!simplified)
541         for(int i=0; i<t->nodesLen; i++)
542           simplify(t->nodes[i]);
543     }
544   }
```

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`.

3.2.7 Funzioni `gcd` e `lcm`

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

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

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[7] 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.

[7] **euclid**.

3.2.8 Funzione `getLCM`

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

`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,[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:

[8] **mcmt**.

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

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.

3.2.9 Funzione `normalize`

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

```
220
221       else if (strcmp(">", tree->nodes[i]->nodeName) == 0) {
222         t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
223
224         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
225           if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0)
226             c = atoi(addends[j]->nodes[0]->nodeName);
227         }
228
229         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
230           if (strcmp(addends[j]->nodes[1]->nodeName, var) == 0) {
231             if(c > 0) strcpy(addends[j]->nodeName, "");
232             else strcpy(addends[j]->nodeName, "-");
233             strcat(addends[j]->nodeName, var);
234             free(addends[j]->nodes[0]);
235             free(addends[j]->nodes[1]);
236             addends[j]->nodesLen = 0;
237           }
238           else {
239             sprintf(addends[j]->nodes[0]->nodeName,
240                     "%d",
241                     atoi(addends[j]->nodes[0]->nodeName)*lcm/abs(c));
242           }
243         }
244
245         sprintf(tree->nodes[i]->nodes[1]->nodeName,
246                 "%d",
247                 atoi(tree->nodes[i]->nodes[1]->nodeName)*lcm/abs(c));
248       }
249     }
250
251     tree->nodesLen++;
252     tree->nodes = realloc(tree->nodes, sizeof(t_syntaxTree*) * tree->nodesLen);
253     tree->nodes[tree->nodesLen-1] = malloc(sizeof(t_syntaxTree));
254     strcpy(tree->nodes[tree->nodesLen-1]->nodeName, "div");
255     tree->nodes[tree->nodesLen-1]->nodesLen = 2;
256     tree->nodes[tree->nodesLen-1]->nodes = malloc(sizeof(t_syntaxTree*) * 2);
257     tree->nodes[tree->nodesLen-1]->nodes[0] = malloc(sizeof(t_syntaxTree));
258     tree->nodes[tree->nodesLen-1]->nodes[1] = malloc(sizeof(t_syntaxTree));
259     tree->nodes[tree->nodesLen-1]->nodes[0]->nodesLen = 0;
260     tree->nodes[tree->nodesLen-1]->nodes[0]->nodes = NULL;
261     tree->nodes[tree->nodesLen-1]->nodes[1]->nodesLen = 0;
262     tree->nodes[tree->nodesLen-1]->nodes[1]->nodes = NULL;
263     strcpy(tree->nodes[tree->nodesLen-1]->nodes[0]->nodeName, var);
264     sprintf(tree->nodes[tree->nodesLen-1]->nodes[1]->nodeName, "%d", lcm);
265   }
```

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

3.2.10 Funzione `minInf`

```
297 t_syntaxTree* minInf(t_syntaxTree* tree, char* var) {
298   t_syntaxTree* nTree = recCopy(tree);
299
300   char minvar[16];
301   minvar[0] = '\0';
302   strcpy(minvar, "-");
303   strcat(minvar, var);
304
305   for (int i=0; i<nTree->nodesLen; i++) {
306     if (strcmp(">", nTree->nodes[i]->nodeName) == 0) {
307       t_syntaxTree** addends = nTree->nodes[i]->nodes[0]->nodes;
308
309       for (int j=0; j<nTree->nodes[i]->nodes[0]->nodesLen; j++) {
310         if (strcmp(addends[j]->nodeName, var) == 0)
311           strcpy(nTree->nodes[i]->nodeName, "false");
312         else if (strcmp(addends[j]->nodeName, minvar) == 0)
313           strcpy(nTree->nodes[i]->nodeName, "true");
314       }
315
316       for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
317         recFree(nTree->nodes[i]->nodes[j]);
318
319       free(nTree->nodes[i]->nodes);
320       nTree->nodes[i]->nodesLen = 0;
321       nTree->nodes[i]->nodes = NULL;
322     }
323
324     else if (strcmp("=", nTree->nodes[i]->nodeName) == 0) {
325       for (int j=0; j<nTree->nodes[i]->nodesLen; j++)
326         recFree(nTree->nodes[i]->nodes[j]);
327
328       free(nTree->nodes[i]->nodes);
329       nTree->nodes[i]->nodesLen = 0;
330       nTree->nodes[i]->nodes = NULL;
331       strcpy(nTree->nodes[i]->nodeName, "false");
332     }
333   }
334
335   return nTree;
336 }
```

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

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

3.2.11 Funzione `calcm`

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

  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.

3.2.12 Funzione `boundaryPoints`

```
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;
```

```
393             strcpy(bp->nodeName, "+");
394             bp->nodesLen = 0;
395 
396             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
397               if (strcmp(var, addends[k]->nodeName) != 0) {
398                 bp->nodesLen++;
399                 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
400                 bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
401                 sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
402                 strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
403               }
404             }
405 
406             bp->nodesLen++;
407             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
408             bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
409             bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
410             bp->nodes[bp->nodesLen - 1]->nodes = NULL;
411             sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
412             strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);
413 
414             bPoints->nodes[bPoints->nodesLen-1] = bp;
415             break;
416           }
417         }
418       }
419 
420       if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
421         t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
422 
423         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
424           if (strcmp(var, addends[j]->nodeName) == 0) {
425             bPoints->nodesLen++;
426             bPoints->nodes = realloc(bPoints->nodes, sizeof(t_syntaxTree *) * bPoints->nodesLen);
427             t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
428             bp->nodes = NULL;
429             strcpy(bp->nodeName, "+");
430             bp->nodesLen = 0;
431 
432             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
433               if (strcmp(var, addends[k]->nodeName) != 0) {
434                 bp->nodesLen++;
435                 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
436                 bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
437                 sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
438                 strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
439               }
440             }
441 
442             bp->nodesLen++;
```

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

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`.

3.2.13 Funzione `newFormula`

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

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

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.

3.2.14 Funzione `eval`

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

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.

4.1 Il programma `test.c`

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

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

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

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

  free(str);

  return 0;
}
```

Si consideri ora il seguente `makefile`:

4.2 Il Makefile

```
SHELL := /bin/bash
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

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

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

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

É 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]

4.3 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)
13
14 (define (main args)
15   (let ((str (cadr args)))
16     (format #t
17             "\nInput: ~s\nEvaluated: ~s\n"
```

[9] **guile**.

```
            str
            (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 soddsfacibilità si usi invece la regola `sat` del `makefile`. Tale regola esegue il seguente script Python.[10]

```python
#!/bin/python3
from sys import argv
from subprocess import run


def main():
    if len(argv) != 4:
        print("Wrong arguments number!")
    else:
        wff = argv[1]
        variables = argv[2].split()
        var = argv[3]
        yices = ""

        for var in variables:
            yices += "(define {}::int)\n".format(var)

        wff_out = run(["./test", wff, var], capture_output=True).stdout.decode()
        yices += "(assert {})\n(check)".format(wff_out)

        with open("source.ys", "w") as source:
            print(yices, file=source)

        run(["yices", "source.ys"])


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

Tale script genera un opportuno sorgente `source.ys` per Yices[11] 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"`.

```
(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))
```

[10] **python**.
[11] **yices**.

```
            (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