Algoritmo di eliminazione dei quantificatori di Cooper

una semplice implementazione scritta in linguaggio C

Andrea Ciceri

21 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 $s < t + 1$).

[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:2050/10.1080/01445340910883718 7`.

- 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 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.

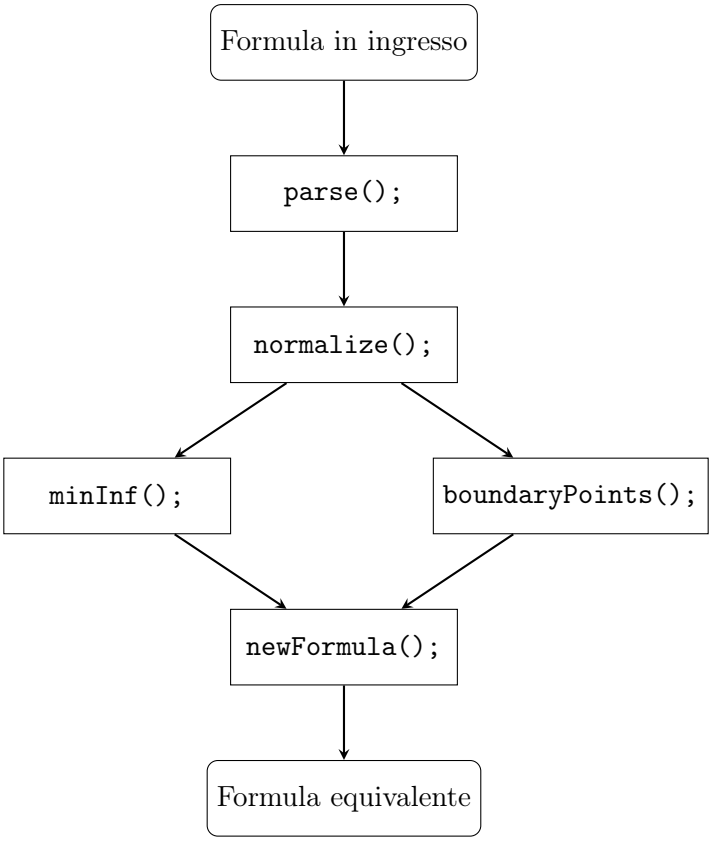
[3] D. C. Cooper. "Theorem proving in arithmetic without multiplication". In: *Machine Intelligence 7* (1972), pp. 91–99. URL: `http://citeseerx.ist.psu.edu/showciting?cid=697241`.

[4] Michael J. Fischer e Michael O. Rabin. "Super-Exponential Complexity of Presburger Arithmetic". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. A cura di Bob F. Caviness e Jeremy R. Johnson. Vienna: Springer Vienna, 1998, pp. 122–135. ISBN: 978-3-7091-9459-1.

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.

[5] 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`.

[6] Clark Barrett and Pascal Fontaine and Cesare Tinelli. *SMT-LIB*. ver. 2.6. 18 Giu. 2017. URL: `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf`.

Il passo successivo al parsing è la normalizzazione della formula, cioè la generazione della formula $\varphi' = D_l(x) \wedge \psi$, dove i coefficienti della variabile da eliminare sono diventati 1. La segnatura di tale funzione è `void normalize(t_syntaxTree* tree, char* var)`.

Le funzioni `t_syntaxTree* minInf(t_syntaxTree* tree, char* var)` e `t_syntaxTree* boundaryPoints(t_syntaxTree* tree, char* var)`, come è facile evincere, generano rispettivamente $\varphi'_{-\infty}$ e l'insieme dei boundary points.

Infine `t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var)` genera la formula equivalente a partire da $\varphi'_{-\infty}$ e 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`

```
587 char* cooperToStr(char* wff, char* var) {
588   t_syntaxTree* tree, *minf, *f;
589   char* str;
590
591   tree = parse(wff); //Genera l'albero sintattico a partire dalla stringa
592   normalize(tree, var); //Trasforma l'albero di tree
593   minf = minInf(tree, var); //Restituisce l'albero di φ_-∞
594   f = newFormula(tree, minf, var); //Restituisce la formula equivalente
595   //simplify(f); //opzionale
596   str = treeToStr(f); //Genera la stringa a partire dall'albero
597
598   recFree(tree); //Libera la memoria
599   recFree(minf);
600   recFree(f);
601
602   return str;
603 }
```

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`.

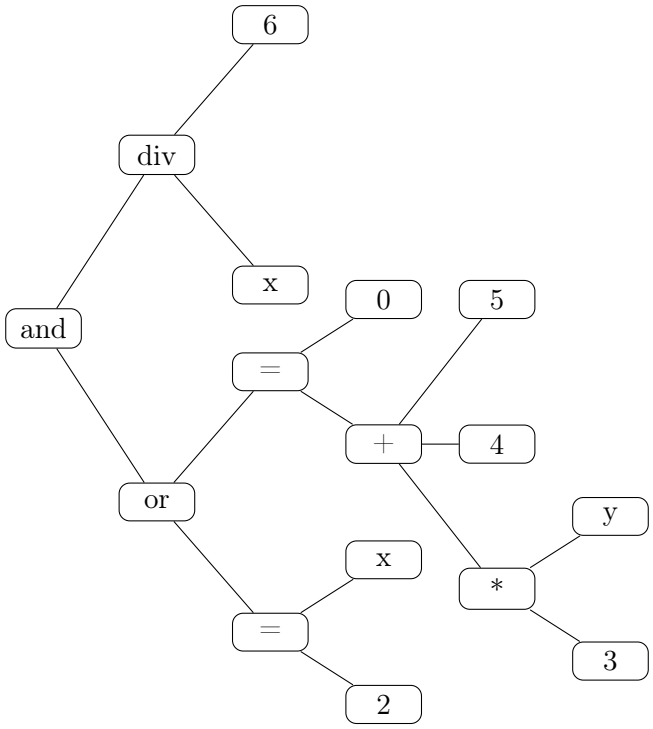
3.2.2 Segnatura di `t_syntaxTree`

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

Trattasi di un record definito 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 $((2 = x) \land (3y + 4 + 5 = 0)) \lor (x \equiv_6 0)$, 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`

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

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

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

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

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

3.2.5 Funzione `treeToStr`

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

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.

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

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

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

```
525       }
526
527       if (strcmp(t->nodeName, "or") == 0) {
528         for(int i=0; i<t->nodesLen; i++) {
529           if (strcmp(t->nodes[i]->nodeName, "true") == 0) {
530             simplified = 1;
```

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`

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

```
531
532             for (int j=0; j<t->nodesLen; j++)
533               recFree(t->nodes[j]);
534
535             strcpy(t->nodeName, "true");
536             t->nodesLen = 0;
537             break;
538           }
539         }
540       }
541
542       if (!simplified)
543         for(int i=0; i<t->nodesLen; i++)
544           simplify(t->nodes[i]);
545     }
546   }
```

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`

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

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

Come è facile immaginare tali funzioni effettuano semplicemente il calcolo del massimo comun divisore e del minimo comune multiplo. Il primo viene svolto efficacemente dall'algoritmo di 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. *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.

3.2.8 Funzione `getLCM`

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

`getLCM` visita ogni nodo dell'albero alla ricerca dei coefficienti della variabile `var`, ovvero cerca nodi della forma `(* c var)` dove appunto `var` è la variabile da eliminare mentre `c` è il coefficiente. 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] 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.

3.2.9 Funzione `normalize`

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

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

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`

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

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`

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

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`

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

```
395             strcpy(bp->nodeName, "+");
396             bp->nodesLen = 0;
397 
398             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
399               if (strcmp(var, addends[k]->nodeName) != 0) {
400                 bp->nodesLen++;
401                 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
402                 bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
403                 sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
404                 strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
405               }
406             }
407 
408             bp->nodesLen++;
409             bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
410             bp->nodes[bp->nodesLen-1] = malloc(sizeof(t_syntaxTree));
411             bp->nodes[bp->nodesLen - 1]->nodesLen = 0;
412             bp->nodes[bp->nodesLen - 1]->nodes = NULL;
413             sprintf(str, "%d", -1+atoi(tree->nodes[i]->nodes[1]->nodeName));
414             strcpy(bp->nodes[bp->nodesLen - 1]->nodeName, str);
415 
416             bPoints->nodes[bPoints->nodesLen-1] = bp;
417             break;
418           }
419         }
420       }
421 
422       if (strcmp(tree->nodes[i]->nodeName, ">") == 0) {
423         t_syntaxTree** addends = tree->nodes[i]->nodes[0]->nodes;
424 
425         for (int j=0; j<tree->nodes[i]->nodes[0]->nodesLen; j++) {
426           if (strcmp(var, addends[j]->nodeName) == 0) {
427             bPoints->nodesLen++;
428             bPoints->nodes = realloc(bPoints->nodes, sizeof(t_syntaxTree *) * bPoints->nodesLen);
429             t_syntaxTree* bp = malloc(sizeof(t_syntaxTree));
430             bp->nodes = NULL;
431             strcpy(bp->nodeName, "+");
432             bp->nodesLen = 0;
433 
434             for (int k=0; k<tree->nodes[i]->nodes[0]->nodesLen; k++) {
435               if (strcmp(var, addends[k]->nodeName) != 0) {
436                 bp->nodesLen++;
437                 bp->nodes = realloc(bp->nodes, sizeof(t_syntaxTree*) * bp->nodesLen);
438                 bp->nodes[bp->nodesLen-1] = recCopy(addends[k]);
439                 sprintf(str, "%d", -atoi(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName));
440                 strcpy(bp->nodes[bp->nodesLen-1]->nodes[0]->nodeName, str);
441               }
442             }
443 
444             bp->nodesLen++;
```

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

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`

```
463 t_syntaxTree* newFormula(t_syntaxTree* tree, t_syntaxTree* minf, char* var) {
464   int m = calcm(minf, var);
465   t_syntaxTree* val;
466   char str[16];
467   t_syntaxTree* nTree = malloc(sizeof(t_syntaxTree));
468   strcpy(nTree->nodeName, "or");
469   nTree->nodesLen = 0;
470   nTree->nodes = NULL;
471
472   t_syntaxTree* t;
473   t_syntaxTree* bp;
474   t_syntaxTree *bPts = boundaryPoints(tree, var);
475
476   for(int i=1; i<=m; i++) {
477     nTree->nodesLen++;
478     nTree->nodes = realloc(nTree->nodes, sizeof(t_syntaxTree *) * nTree->nodesLen);
479     t = recCopy(minf);
480     val = malloc(sizeof(t_syntaxTree));
481     sprintf(str, "%d", i);
482     strcpy(val->nodeName, str);
483     val->nodesLen = 0;
484     val->nodes = NULL;
485     eval(t, var, val);
486     recFree(val);
487     nTree->nodes[nTree->nodesLen-1] = t;
488
489     for(int j=0; j<bPts->nodesLen; j++) {
```

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

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`

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

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] GNU. *GNU Ubiquitous Intelligent Language for Extensions (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 Software Foundation. *Python language*. Ver. 3.7.2. 2019. URL: `https://www.python.org/`.
[11] SRI International. *Yices*. Ver. 1.0.40. 4 Dic. 2013. URL: `http://yices.csl.sri.com/`.

```
            (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

Riferimenti bibliografici

Clark Barrett and Pascal Fontaine and Cesare Tinelli. *SMT-LIB*. Ver. 2.6. 18 Giu. 2017. URL: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

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

Euclid. *Euclid's Elements*. All thirteen books complete in one volume, The Thomas L. Heath translation, Edited by Dana Densmore. Green Lion Press, Santa Fe, NM, 2002, pp. xxx+499. ISBN: 1-888009-18-7; 1-888009-19-5.

Fischer, Michael J. e Michael O. Rabin. "Super-Exponential Complexity of Presburger Arithmetic". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. A cura di Bob F. Caviness e Jeremy R. Johnson. Vienna: Springer Vienna, 1998, pp. 122–135. ISBN: 978-3-7091-9459-1.

Ghilardi, Silvio. *MCMT: Model Checker Modulo Theories*. http://users.mat.unimi.it/users/ghilardi/mcmt/. 2018.

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

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:2050/10.1080/01445340910883718 7.

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

SRI International. *Yices*. Ver. 1.0.40. 4 Dic. 2013. URL: http://yices.csl.sri.com/.