Implementazione di una procedura di soddisfacibilità per la teoria delle liste

Enrico Scapin vr353597

21 luglio 2011

1 Introduzione

Nel processo di verifica automatica di programmi le procedure di soddisfacibilità sono essenziali in quanto, dopo una prima fase di annotazione del codice, è necessario disporre di un *Dimostratore Automatico di Teoremi* che assicuri la validità di ciascuna condizione di verifica generata.

Le condizioni di verifica sono espresse tramite un alfabeto Σ che denota una od un insieme di teorie: queste teorie formalizzano le strutture usualmente manipolate dai programmi (come numeri, array, liste) permenttendo così un ragionamento automatico su di esse.

Il dimostratore automatico di teoremi procede in maniera refutazionale sul frammento senza quantificatori di ogni condizione di verifica: infatti una formula è valida se e solo se la sua negazione è insoddisfacibile. Per questa ragione è necessario disporre di procedure di soddisfacililità basate sul rispetto degli assiomi presenti in ciascuna teoria.

In questo progetto è stata implementata la procedura di soddisfacibilità per la teoria delle liste T_{cons} sulla base dell'algoritmo di chiusura di congruenza sviluppato da G. Nelson e D. C. Oppen. Esso opera mediante l'ausilio di un *Directed Acyclic Graph* (DAG) in cui ogni nodo rappresenta un elemento dell'insieme dei sottotermini della formula, mentre ogni arco collega un sottotermine con i suoi argomenti. La complessità di tale algoritmo è quadratica sul numero di archi del grafo, $O(e^2)$.

2 Scelte implementative

L'elaborato è stato implementato utilizzando il linguaggio *Java* in quanto il livello di astrazione è tale da permettere al programmatore di concentrarsi principalmente sulla progettazione dell'algoritmo ed, in particolare, su quali strutture dati sia meglio utilizzare. Inoltre la scelta di Java consente l'utilizzo di *JavaCC*, uno strumento che permette di effettuare il parsing delle formule semplicemente specificando l'alfabeto e la grammatica del linguaggio.

Poiché si è deciso di mantenere il più possibile separate la fase di parsing da quella di implementazione, è stato necessario creare una classe "contenitore", `CCObject` (*src/bean*), in grado di mantenere sia i riferimenti a tutte le strutture dati create dal parser, sia alcune informazioni visualizzate in output (e.g. il numero di archi).

Per quanto riguarda le strutture dati, si è fatto un ingente uso di tavole hash tramite le classi `HashMap` ed `HashSet`; dove invece si è reso necessario mantenere l'ordine di inserimento, è stata utilizzata le struttura dati delle liste, tramite la classe `LinkedList`.

La scelta di utilizzare diffusamente tavole hash è avvalorata dal fatto che queste strutture, oltre a supportare le operazioni `insert`, `contains` e `remove` in tempo atteso costante, prevengono efficacemente la presenza di oggetti doppi al loro interno. In particolare il DAG è implementato tramite un `HashMap` le cui chiavi sono i campi `id` dei vari nodi. Ciò permette di non dover mantenere, nelle altre strutture di supporto, il riferimento direttamente a tali nodi ma solamente le loro chiavi, sapendo che poi la ricerca all'interno del DAG avviene in maniera efficiente.

2.1 Parser

Dopo che la stringa è stata letta (inserita tramite riga di comando oppure prelevata da un file), il parser si preoccupa di farne un'analisi lessicale e sintattica per il riconoscimento di formule espresse nell'alfabeto $\Sigma_E \cup \Sigma_{cons}$, ovvero l'unione delle segnature della teoria dell'uguaglianza e delle liste (che ne è un estensione).

Il parser, che si può trovare nella cartella *src/parser*, consiste in un file con estensione `.jj` costituito da un'unità di compilazione java e da una grammatica context-free di tipo *LL(2)* le cui produzioni sono espresse in BNF (*Backus–Naur Form*). La grammatica è formata dalle seguenti categorie sintattiche:

```
Formula ::= Clause (; Clause? )*
Clause  ::= Term = Term | Term != Term | atom(Term) | -atom(Term) |
            (-)? Pred(Term (, Term)*)
Term    ::= cons(Term, Term) | car(Term) | cdr(Term) | FunVar(Term (, Term)*) | FunVar
```

I token `Pred` e `FunVar` sono specificati dalle seguenti espressioni regolari:

```
Pred    ::= ([A-Z])+([0-9])*
FunVar  ::= ([A-Z])*([a-z])+([a-z,A-Z])*([0-9])*
```

Considerazioni:

- il punto e virgola denota una congiunzione tra clausole;
- sono stati utilizzati due caratteri diversi per la negazione solamente per ovviare ad un problema di compatibilità con la bash in caso l'input sia inserito da riga di comando;
- i simboli di predicato non interpretati sono espressi tramite lettere maiuscole mentre i simboli di funzione e di variabile con almento una lettera minuscola (opzionalmente possono anche essere seguiti da cifre);
- ogni clausola identifica un unico predicato che è formato da uno o più termini; ogni termine può, a sua volta, essere formato da sottotermini oppure identificare un simbolo terminale, cioé una variabile o una costante (nel frammento senza quantificatori si perde la distinzione fra queste due categorie sintattiche).

Per quanto riguarda l'aspetto semantico, per ogni predicato e termine è stato inserito del codice Java che principalmente si occupa della costruzione del DAG, mantenuto tramite una tavola hash implementata utilizzando la classe `HashMap`. Il grafo è costruito mediante un approccio bottom-up.

- Ogni produzione di ciascun termine riceve uno o più oggetti di tipo `Node` (*src/bean*) che rapprensentano i nodi del grafo dei suoi argomenti; viene quindi creato un nuovo nodo rappresentante quel termine ed si effettua il matching tra il suo campo *argument* (`LinkedList`) ed il campo *parent* (`HashSet`) dei suoi figli. Ciò avviene inserendo la chiave del nodo nei campi *parent* dei figli e le chiavi dei figli nel campo *argument* del padre.
- Ogni produzione di ciascun predicato riceve anch'esso uno o più oggetti di tipo `Node` ma si comporta in maniera diversa a seconda del tipo di predicato:
  a) per quanto riguarda i predicati di uguaglianza, vengono semplicemente inserite le coppie di termini[1] in due insiemi a seconda che il predicato sia positivo o negativo;
  b) per quanto riguarda il prediato `atom`, se è positivo il suo argomento viene inserito nell'insieme (`HashSet`) di tutti i termini che sono argomenti di `atom`, mentre se è negativo si effettua una trasformazione sintattica creando tre nuovi nodi: due variabili fresh ed un termine `cons` che ne diviene il padre; la coppia formata da questo nuovo nodo e dall'argomento del predicato viene poi aggiunta all' insieme dei predicati di uguaglianza;
  c) per quanto riguarda gli altri predicati non interpretati si effettua anche in questo caso una trasformazione sintattica in maniera da convertirli in funzioni; li si inserisce poi nell'insieme dei predicati di uguagliaza o di disuguaglianza, a seconda che siano positivi o negativi, in coppia con un simbolo speciale che è stato scelto essere "#".

Per quanto riguarda le produzioni delle funzioni e dei predicati in cui il numero di argomenti non è noto a priori, si mantiene una corrispondenza (`HashMap`) tra i simboli letti ed il corrispettivo numero di argomenti. In questo modo, in caso si trovino due simboli uguali con un diverso numero di argomenti, viene sollevata un'eccezione.
Durante l'elaborazione di ogni nodo, vengono aggiornati anche tutti quei dati, come il numero di archi o di clausule di uguaglianza, che saranno poi forniti in output.

2.2 Algoritmo di Nelson-Oppen

Dopo che è stato effettuato il parsing dell'intera formula e creato il DAG, viene eseguito l'algoritmo di chiusura di congruenza di cui ne sono state implementate due versioni selezionabili dall'utente mediante l'eventuale uso dell'opzione `-h`.

La prima versione è la semplice trasposizione in codice Java dell'algoritmo presente al capitolo 9.3 del libro *The Calculus of Computation* di A. R. Bradley e Z. Manna.

[1] Poichè si è reso necessario memorizzare i termini in coppia, è stato creata la classe `TermPair` (*src/bean*) che tramite l'overriding dei metodi `hashCode` ed `equals` permette di mantenere coppie non ordinate nei vari `HashSet`.

La seconda versione è una nuova reingegnerizzazione di tale algortimo tramite l'utilizzo di alcune euristiche atte soprattutto a permettere, non appena viene trovata una contraddizione tra due termini, l'immediata interruzione del flusso di esecuzione restituendo insoddisfacibile.

L'idea è quella di mantenere per ogni nodo un *banned set* composto da tutti quei termini che non possono appartenere alla sua stessa classe di congruenza. Tale insieme viene popolato durante il parsing: ogniqualvolta si incontra un predicato di disuguaglianza $s_i \neq t_i$, viene aggiunto al *banned set* del nodo che rappresenta s_i il termine t_i e viceversa. In questo modo ogni volta che due termini devono essere uniti in base agli assiomi di congruenza, si controlla se uno è nel *banned set* dell'altro ed, in caso affermativo, si restituisce immediatamente insoddisfacibile.

Questa euristica può essere ulteriormente migliorata sulla base di questa considerazione: quando si effettua l'operazione di `merge` tra due nodi in realtà si stanno unendo le loro classi di congruenza e quindi è utile controllare non solo se i due termini formano un predicato di disuguaglianza, ma anche se nel *banned set* di ciausun nodo vi è il riferimento ad uno dei nodi appartenenti alla stessa classe di congruenza dell'altro. Per implementare ciò, quando si effettua l'operazione di `union`, non si fa l'unione solamente tra i campi `ccpar`, ma anche tra i campi `banned`. In questo modo, ogniqualvolta viene eseguito il metodo `merge` tra due nodi, si controlla se è possibile unire le loro classi di congruenza: si seleziona il *banned set* del rappresentante di uno dei due e, per ogni termine presente in esso, si controlla se la sua classe di congruenza è la stessa classe dell'altro nodo con cui si vuole effettuare la `union` ritornando, in caso positivo, insoddisfacibile.

```
// id1, id2 : nodes' id for which make union
for(String ban: node(find(id1)).getBanned())
                if(find(ban).equals(find(id2))) // CONFLICT
                        return new TermPair(id1, id2);
```

Tale operazione è ripetuta simmetricamente anche per l'altro nodo e non peggiora la complessità dell'algoritmo in quanto è lineare sul numero di nodi dei *banned set*.

Sulla base di questa euristica, che in pratica modifica i *banned set* durante l'applicazione degli assiomi di congruenza, possiamo riformulare l'algoritmo di Nelson-Oppen.

1. Controllare che nessun argomento dei predicati `atom` sia un termine `cons`, in accordo con l'assioma *(atom)* della teoria delle liste. Se ne viene rilevata la presenza restituire UNSAT.
2. Aggiungere ai *banned set* dei nodi che rappresentano gli argomenti del predicato `atom` tutti i termini `cons` (e viceversa) in modo da rispettare l'assioma precedente anche durante la propagazione degli effetti di nuove congruenze.
3. Per ogni nodo `cons` aggiungere al DAG due nuovi nodi `car` e `cdr` i cui archi puntano a quel nodo e poi effettuare la `merge` del `car` con il primo argomento del `cons` e del `cdr` con il secondo. In caso di conflitto durante la `merge`, restituire UNSAT.
4. Effettuare la `merge` di ogni coppia di termini presente nell'insieme dei predicati di uguaglianza. In caso di conflitto restituire UNSAT.
5. Ritornare SAT.

Da notare che in realtà nell'implementazione l'algoritmo non restituisce UNSAT bensì la coppia di termini tra i quali si è manifestato tale conflitto. Inoltre la percentuale che si può osservare durante l'esecuzione, consiste nella percentuale del numero di clausole di uguaglianza fino a quell'istante computate.

Poichè l'algoritmo opera sui nodi del DAG come su una foresta di insiemi disgiunti, è possibile implementare altre due euristiche nei metodi `find` ed `union` in modo da abbassarne ulteriormente il tempo di esecuzione.

- **Path Compression**: la compressione dei cammini, implementata nel metodo `find_h`, consiste nell'appiattire la stuttura dell'albero che identifica ciascuna classe di congruenza facendo puntare tutti i nodi direttamente al rappresentante; in questo modo le successive chiamate a tale funzione su quei nodi possono essere espletate in tempo costante. Per ottenere ciò si applica un metodo a doppio passaggio: durante il primo passaggio, il metodo risale ricorsivamente il cammino di ricerca per trovare la radice; durante il secondo passaggio, ridiscende il cammino di ricerca per aggiornare i nodi in modo che puntino direttamente al rappresenante.
- **Union by Rank**: l'unione per rango, implementata nel metodo `union_h`, consiste nell'unire due classi di congruenza in modo che il nuovo rappresentante sia quello della classe più grande delle due. Il vantaggio è che i nodi della classe di congruenza che cambia rappresentante avranno una distanza maggiore dalla radice e quindi l'operazione di `find` sarà più onerosa. Per implementare ciò, ogni nodo è provvisto di un campo `rank` che è un limite superiore al numero di archi tra quel nodo e una foglia discendente. Quando due classi di congruenza vengono unite, si trasforma il rappresentante di rango più alevato nel rappresentante di rango più basso, senza però modificare questo campo. Se

invece i ranghi dei due rappresentanti sono uguali, se ne trasforma arbitrariamente uno nel padre dell'altro, incrementandone anche il rango.

3 Benchmark

In questa sezione viene riportata l'analisi delle prestazioni di tale algoritmo effettuata su un calcolatore con CPU `Intel Core 2 Duo P8600 2.4GHz`, sistema operativo `Ubuntu 10.10` e `JVM v1.6.0_24`.
Le formule sono state create tramite un generatore automatico, `formulaGenerator`, che fa un ingente uso di una funzione random. Inoltre per automatizzare i test è stato implementato uno script (`generatorTest.sh`) che crea e salva i risultati in un file denominato `testResult.cvs`.

Di seguito sono presentate due tabelle con alcuni dei dati raccolti sulle prestazioni dell'algoritmo: la prima riguarda i test effettuati su formule composte da clausole miste, mentre la seconda su formule contenenti solo clausole di uguaglianza, `-atom` e predicati positivi che le rendono quindi sicuramente soddisfacibili.

Clausole	Nodi	Archi	Risultato	Tempo (ms)	Tempo eur (ms)
64	415	1162	UNSAT	43	56
384	1417	3809	UNSAT	14841	149
512	1747	4926	UNSAT	8407	251
768	2959	8359	UNSAT	40988	422
1152	3581	10563	UNSAT	78023	328
1408	6514	22319	UNSAT	997942	924

Tabella 1: Prestazioni dell'algoritmo su formule con clausole miste

Clausole	Nodi	Archi	Risultato	Tempo (ms)	Tempo eur (ms)
64	427	1357	SAT	86	77
512	2018	6111	SAT	21660	1598
384	2381	8450	SAT	22064	2427
1152	4108	11633	SAT	153478	5441
768	4243	15041	SAT	124388	7663
1408	8582	31080	SAT	1184730	34355

Tabella 2: Prestazioni dell'algoritmo su formule soddisfacibili

3.1 Considerazioni

Nelle tabelle sopra riportate si è deciso di ordinare le righe a seconda del numero di archi nel grafo visto che è su questa grandezza che viene calcolata la complessità. Dalla seconda tabella si può notare che ad un aumento del numero di clausole non sempre corrisponde un proporzionale aumento del numero di archi: infatti il numero di nodi e di archi dipende principalmente dalla quantità di sottotermini presenti nella formula e questa grandezza varia in maniera casuale per ogni formula generata.

Dalle tabelle sopra riportate si può facilmente notare che, quando il numero di archi è limitato, l'algoritmo originale, non dovendo compiere attività di preprocessing (e.g. scorrere ed aggiornare i vari *banned set*) riescie ad essere più performante dell'algoritmo con euristiche come nel caso del primo test della prima tabella.

Quando invece la grandezza del DAG aumenta, tramite le euristiche si ottengono migliori performance: nei test di formule con clausole miste, poiché il risultato è sempre UNSAT, la differenza tra le prestazioni è dovuta principalmente all'euristica che fa uso dei *banned set* per restituire insoddisfacibile non appena trova un conflitto. Dalla prima tabella si può notare anche che, grazie a quest'euristica, il tempo di esecuzione non è sempre direttamente proporzionale al numero di archi ma esso dipende piuttosto dall'instante in cui si trova la prima contraddizione.

Nei test di formule soddisfacibili, invece, l'euristica precedente è inefficiente in quanto sicuramente non vi sono contraddizioni tra termini o predicati; in questo caso la differenza di prestazioni è data dall'implementazione, nei metodi `union` e `find`, delle tecniche di unione per rango e compressione dei cammini.