Implementazione di un dimostratore di teoremi per risoluzione

Enrico Scapin vr353597

1 febbraio 2012

1 Introduzione

Uno degli obiettivi del ragionamento automatico consiste nel cercare di costruire *Dimostratori Automatici di Teoremi* al fine di ottenere meccanismi automatizzabili per asserire, a partire da un insieme di assunzioni H, la validità o meno di una determinata congettura φ. Più formalmente:

$$H \models \varphi$$

I dimostratori automatici tipicamente procedono in maniera refutazionale in quanto ogni formula è valida se e solo se la sua negazione è insoddisfacibile. Quindi, riferendoci al sequente sopra:

$$H \models \varphi \iff H \cup \{\neg\varphi\} \text{ è insoddisfacibile}$$

Da questa considerazione possiamo quindi cercare di costruire una procedura che, se dimostra l'insoddisfacibilità di $H \cup \{\neg\varphi\}$, allora $H \models \varphi$ è valido, altrimenti, se ne dimostra la sua soddisfacibilità, il modello che lo soddisfa costituirà il controesempio alla validità di $H \models \varphi$.

Siamo interessati a dimostrare formule espresse in Logica del Primo Ordine[1] in quanto, al contrario della logica proposizionale, essa è sufficientemente espressiva da poter modellare una buona parte della nostra conoscenza. Utilizzando questo linguaggio però il problema della validità non è più decidibile bensì semidecidibile: infatti se l'insieme $H \cup \{\neg\varphi\}$ è insoddisfacibile allora la procedura terminerà sempre con la risposta corretta, mentre se esso è soddisfacibile non è detto che la procedura termini.

La semidecidibilità deriva direttamente dal *Teorema di Herbrand* che afferma che un insieme finito S di clausole in *FOL* è soddisfacibile se e solo se esiste un insieme finito S' di istanze ground (clausole in cui tutte le variabili sono istanziate ad una qualche costante) di clausole di S tale che S' è soddisfacibile. Quindi per dimostrare la soddisfacibilità di S è necessario generare tutti gli insiemi S' di istanze ground e dimostrarne la soddisfacibilità ma, se in S si quantifica su insiemi infiniti, allora la cardinalità degli insiemi S' da generare è anch'essa infinita.

Questo dimostratore prende in ingresso un insieme di clausole scritte in Forma Normale Congiunta[2] e definite da una sintassi standard compatibile con frammento CNF senza uguaglianza della libreria *TPTP* (vedi [4]). Si è quindi implementata una procedura di semi-decisione che, basata su un sistema di cinque regole di inferenza (di cui due di espansione e tre di contrazione), implementa un piano di ricerca denominato *Ciclo della Clausola Data* (Given clause loop) che è uno standard alla base di molti dimostratori di insieme di formule in logica al primo ordine come Otter, E, Vampire, Gandalf etc.

2 Scelte progettuali ed 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.

2.1 Parser

La scelta di *Java* consente inoltre l'utilizzo di *JavaCC*, uno strumento che ha permesso di effettuare sia il parsing delle formule sia quello degli argomenti che vengono passati da riga di comando: i due parser consistono in file con estensione `.jj` costituiti da un'unità di compilazione java e da una grammatica context-free di tipo *LL(1)* le cui produzioni sono espresse in BNF (*Backus–Naur Form*). Per quanto

[1] *FOL*, First Order Logic
[2] *CNF*, Conjunctive Normal Form

riguarda il parsing degli argomenti la grammatica è molto semplice:

```
SelectionStrategy ::= -fifo | -best(<Numeric>)?
SearchStrategy    ::= -contr | -exp
LoopType ::= -o | -e
Time      ::= -time<Numeric>
FilePath ::= <Char>
```

I token `<Numeric>` e `<Char>` sono specificati dalle seguenti espressioni regolari:

```
Numeric ::= ([0-9])+
Char    ::= ^[-,\t]( ^[\t])*
```

Queste categorie indicano le possibilità con coi può essere eseguito il dimostratore implementato e sono documentate nel file README: si è deciso di utilizzare il parser per dare all'utente la possibilità di inserirli nell'ordine preferito ed inoltre solo il token `FilePath` è obbligatorio (in quanto consiste nella formula in input che deve essere dimostratata). La configurazione di default degli argomenti in cui è prevista una scelta è la seguente:

```
SelectionStrategy ::= -best
SearchStrategy    ::= -contr
LoopType ::= -o
```

Il significato semantico di ognuno di essi sarà poi spiegato successivamente.

Per quanto riguarda invece il parsing del file contentente le formule, la grammatica corrisponde al frammento *CNF* senza uguaglianza di quella di TPTP che può essere reperita in [5].

Per quanto riguarda l'aspetto semantico è stato inserito del codice Java che si preoccupa di costruire ciascuna clausola, partendo dai suoi letterali e a loro volta dai loro termini, per poi inserirla nella struttura dati apposita. Il tipo di struttura dati dipende dal comando specificato dall'utente per il token `SelectionStrategy`: nel caso sia stato inserito `-best` la struttura dati è una coda di min priorità in cui l'ordinamento è definito dal numero di simboli che compongono ciascuna clausola, altrimenti nel caso sia stato inserito `-fifo` la struttura dati è una lista in cui le clausole vengono inserite nell'ordine in cui vengono lette.

2.2 Bean Class

La scelta dell'utilizzo di *Java* ha permesso un minimo di progettazione orientata agli oggetti soprattutto delle cosidette classi bean[3] e, in particolare, delle classi che definisco i termini (funzioni, variabili e costanti) e i letterali.

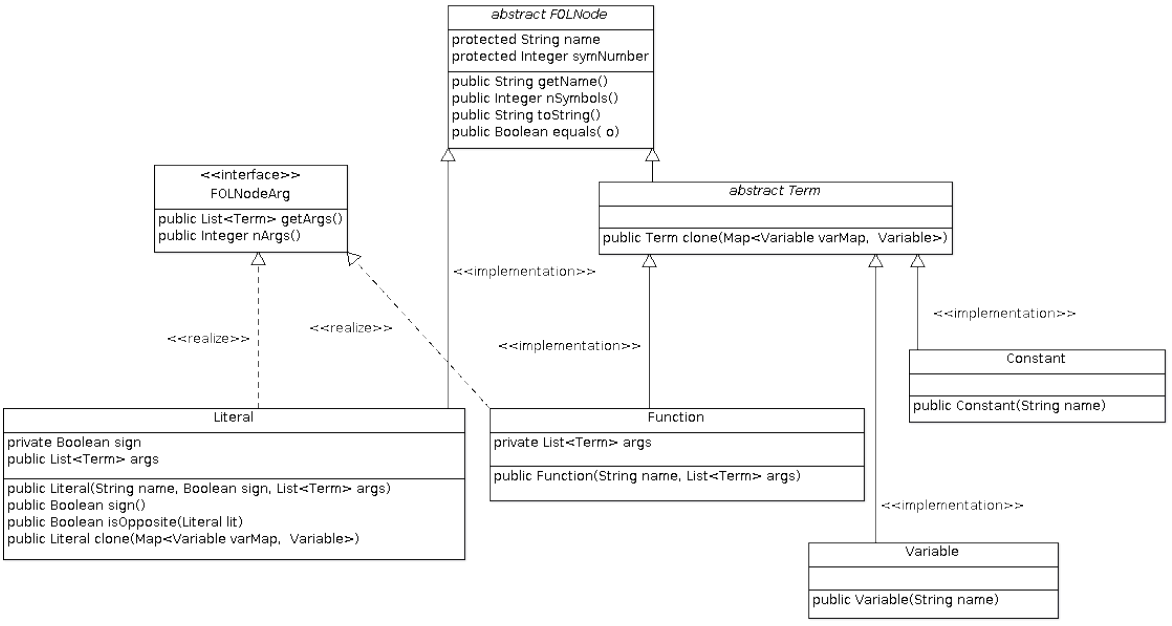

Figura 1: UML delle bean class

[3] così si denotano le classi che logicamente contengono le informazioni da manipolare

Il diagramma UML visibile in Figura 1 rappresenta come è state progettate le classi: ovviamente è molto rilevante la classe astratta `Term`, che rappresenta un qualsiasi oggetto di tipo `Function`, `Variable` o `Constant`, in quanto è spesso necessario riferirsi ad una di esse indistintamente (come ad esempio quando si specificano gli argomenti di un letterale). Il metodo `clone(Map<Variable, Variable>)` presente sia nella classe `Term` sia nella classe `Literal` permette di clonare l'oggetto in questione restituendone un altro con la stessa struttura. Questo metodo si rivela essere molto importante quando è necessario generare una nuova clausola a partire da un'altra: infatti è sufficiente chiamare il metodo `clone` su ogni letterale che ricorsivamente clonerà anche i termini che compongono i suoi argomenti. Quando si crea un nuova clausola è però necessario mantenere inalterato il numero delle variabili in modo tale che tutte le occorrenze di ogni variabile nella vecchia clausola vengano rimpiazzate dalla stessa nuova variabile nella nuova: per far ciò è necessario avere una mappa che mantiene il collegamento tra il vecchio oggetto `Variable` con il nuovo se esso è già stato creato in precedenza in quanto quella determinata variabile è stata già incontrata in precedenza durante la clonazione. Alternativamente se quella variabile non è stata incontrata in precedenza allora è necessario crearne una nuova e inserire la coppia $(vecchia, nuova)$ all'interno della mappa. Se invece al posto di una variabile il metodo clone è chiamato su una costante, essa non verrà clonata (perché il set delle costanti deve rimanere sempre inalterato) bensì verrà ritornata la stessa.

Per quanto riguarda il metodo `toString` è da segnalare l'implementazione nella classe `Variable` in quanto, per differenziare le variabili diverse ma con lo stesso nome, si è scelto di concatenarci gli ultimi 3 caratteri del loro *hash code* codificato in esadecimale. Da notare infine il metodo `equals` che nel caso delle variabili ritorna `true` se e solo se sono lo stesso oggetto mentre nel caso delle costanti ritorna `true` se esse hanno lo stesso nome (e questo è consistente col fatto che le costanti sono le stesse per tutto l'insieme di clausole). Di conseguenza i metodi `equals` delle funzioni e dei letterali, dopo aver controllato l'uguaglianza dei segni e/o dei nomi, vanno in ricorsione sui termini che compongono i loro argomenti e, solo se sono tutti uguali, ritornano `true`.

2.3 Core Class

Le classi contenute nel package *core* sono quelle responsabili della parte computazionale.

2.3.1 Unifier

La classe Unifier contiene i metodi `findMGU` e `findLeftSubst` che, date due liste di argomenti, restituiscono il primo l'unificatore più generale[4] tra le due liste mentre il secondo una sostituzione che, se applicata alla prima lista di argomenti, li rende sintatticamente identici alla seconda lista. In entrambi i casi essi restitutisco una `HashMap` formata da coppie `<Variable, Term>` se l'MGU o la sostituzione esiste, altrimenti `null`. Per quanto riguarda l'MGU si è deciso di implementare l'algoritmo presente nel libro *Artificial Intelligence: A Modern Approach* (vedi [2] e [6]) che è quadratico nella grandezza delle espressioni che devono essere unificate. A questo algoritmo è stato aggiunto un metodo chiamato `cascadeSubst` che rende la sostituzione idempotente, ovvero tutte le variabili presenti nel suo range e che fanno parte anche del suo dominio vengono sostituite con il corrispettivo termine.

2.3.2 Substitution

La classe Substitution si occupa data una sostituzione di applicarla ad un letterale o ad un termine. Questi metodi ritornano sempre un nuovo oggetto e non il vecchio a cui è stata applicata la sostituzione in quanto in tutte le regole di inferenza ogni volta che si applica una sostituzione è per generare una nuova clausola che quindi dovrà avere nuovi letterali e termini. Proprio per questa ragione in questa classe si fa ingente uso del metodo `clone` spiegato precedentemente.

2.3.3 ExpansionRules

La classe ExpansionRules implementa le due regole di espansione Risoluzione Binaria e Fattorizzazione entrambe in due modalità differenti: nella prima vengono passate solamente la (o le) clausole e si cerca di generare tutti i fattori (o i risolutori binari) restituendo un oggetto di tipo `Collection` con tutte le nuove clausole generate, mentre nella seconda sono anche passati i due letterali e la regola di inferenza viene applicata una sola volta su di essi. Entrambe le procedure consistono nella verifica della compatibilità trai due letterali, in caso positivo, nella ricerca di MGU tra le due liste dei loro termini e, se questo viene trovato, nella creazione di un nuovo fattore o risolvente binario. A tale scopo sono presenti anche due metodo `createFactor` e `createResolvent` che, chiamando i metodi della classe `Substitution`, creano una nuova clausola.

[4] MGU, Most General Unifier

2.3.4 Clause

La classe Clause, oltre a contenere i metodi per l'aggiunta dei letterali che la compongono, implementa anche le tre regole di contrazione: eliminazione delle tautologie, semplificazione e sussunzione.

Per quanto riguarda l'eliminazione delle tautologie, il metodo `isTautology` due letterali che siano opposti ovvero che abbiano stesso nome, stessi argomenti ma segno opposto: solo in caso positivo questo metodo ritorna `true`.

Per quanto riguarda la semplificazione clausale, il metodo `simplify` prende come argomento un'altra clausola che, se possibile, la semplifica. Per prima cosa si controlla che la clausola inserita abbia un unico letterale, in caso positivo per ogni letterale compatibile nella clausola chiamate si chiama il metodo `findLeftSubst` che cerca una sostituzione da applicare al letterale della clausola inserita in modo che divenga sintatticamente uguale al letterale selezionato. In caso affermativo si elimita dalla clausola chiamate tale letterale. Un'euristica implementata è che questo procedimento non si fermi al primo letterale eliminato ma continui cercando di semplificare più letterali possibile. I letterali eliminati vengono ritornati tramite un oggetto di tipo `Collection`.

Per quanto riguarda la sussunzione, di gran lunga la regola di inferenza più onerosa dal punto di vista computazionale, il metodo `subsumes` controlla se la clausola chiamate sussume la clausola inserita come argomento. Tale metodo implementa l'algoritmo presente nel libro *Symbolic Logic and Mechanical Theorem Proving* (vedi [1]) con qualche euristica. Infatti condizioni necessarie per cui una clausola ne sussuma un'altra sono:

- il numero di letterali della clausola che sussume deve essere minore o uguale al numero di letterali della clausola da sussumere;
- il numero di simboli della clausola che sussume deve essere minore o uguale al numero di simboli della clausola da sussumere: questa condizione è stata dimostrata nel paper *Towards Efficient Subsumption* (vedi [3]);
- per ogni letterale nella clausola che sussume vi deve essere almeno un letterale nella clausola da sussumere con lo stesso nome e stesso segno.

Se queste tre condizioni necessarie sono soddisfatte si procede con l'algoritmo di sussunzione che però è una versione modificata rispetto a quella del libro di testo. Si mantiene sempre l'insieme delle clausole generate U e si termina se viene trovata la clausola vuota oppure se quest'insieme è vuoto. Ciò che cambia è come vengono generate le clausole che ad ogni iterazione vengono inserite:

1. per ogni clausola in U e per ogni letterale di questa clausola viene eseguito il metodo `findLeftSubst` con un letterale della clausola inserita come parametro;
2. in caso si trovi una sostituzione si crea una nuova clausola a partire da quella che era in U a cui si applica la sostituzione senza considerare il letterale che, istanzianto, è sintatticamente equivalente a quello nella clausola inserita come argomento (da notare che ciò è effettuato semplicemente chiamando il metodo `createFactor`);
3. una volta generati tutti le nuove clausole l'insieme U viene svuotato e ripopolato con le nuove clausole generate a quell'iterazione.

Riferimenti bibliografici

[1] Chang C.L. Lee R.C.T. (1973), Symbolic Logic and Mechanical Theorem Proving, *Academic Press.*

[2] Russell S. Norvig P. (2010), Artificial Intelligence: A Modern Approach (Third Edition), *Prentice Hall.*

[3] Tammet T. (1998), *Lecture Notes in Computer Science. Springer Verlag.* Towards Efficient Subsumption

Siti consultati

[4] `http://www.cs.miami.edu/~tptp/`

[5] `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`

[6] `http://aima.cs.berkeley.edu/algorithms.pdf`