Implementazione di un dimostratore di teoremi per risoluzione

Enrico Scapin vr353597

31 gennaio 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 [2]). 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 [3].

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.

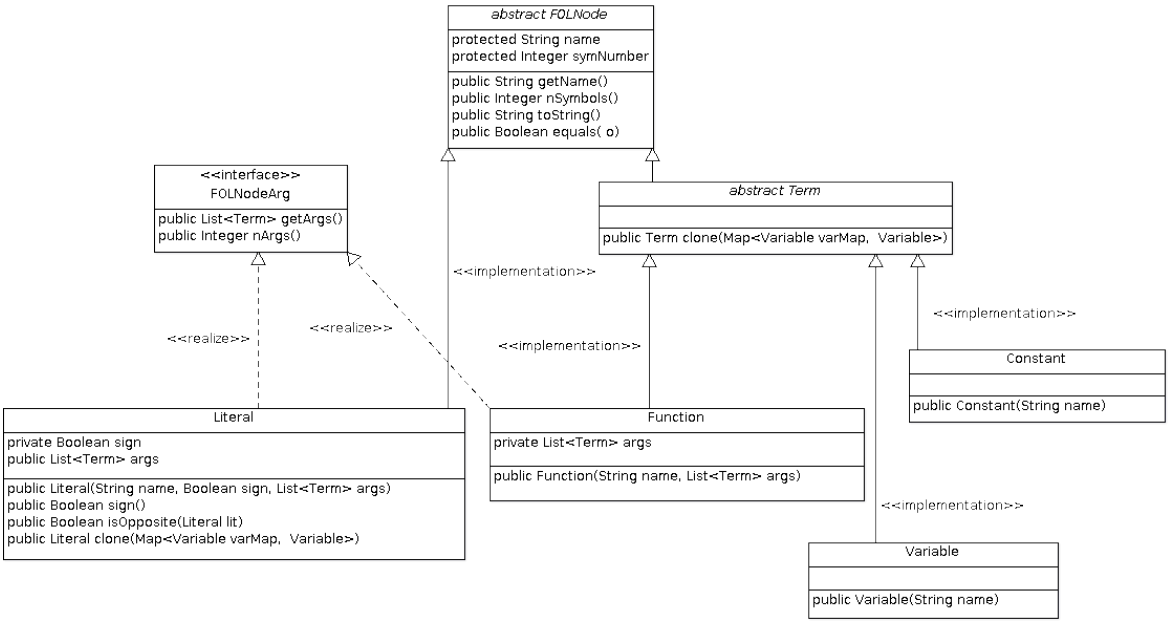

Figura 1: UML delle bean class

[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, andranno in ricorsione sul termini che compongono i loro argomenti e solo se sono tutti uguali ritorneranno `true`.

2.3 Core Class

Riferimenti bibliografici

[1] De Clercq J. (2002), Single Sign On Architettures, *Lecture Notes in Computer Science. Springer Verlag.*

Siti consultati

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

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