Implementazione di un dimostratore basato sul ciclo della clausola data à la Otter e à la E.

Nicolò Marchi - VR365684

4 febbraio 2013

1 Introduzione

In questo progetto per il corso di ragionamento automatico ci siamo occupati di realizzare un dimostratore di teoremi per la risoluzione ordinata basato sul ciclo della clausola data, con sistema di inferenza dato da risoluzione binaria ordinata, fattorizzazione ordinata, eliminazione di tautologie, sussunzione e semplificazione clausale. Il progetto implementa due diversi metodi di ricerca: il metodo à la Otter e il metodo à la E. Si è deciso inoltre di integrare la grammatina fornita dalla libreria standardizzata *TPTP*[7][8], che raccoglie un enorme quantita di problemi già formalizzati in diverse grammatiche, anche se per i nostri scopi ci siamo concentrati solo sulla forma clausale normale, detta CNF (*Clausal Normal Form*). Infatti questo dimostratore prende in ingresso un insieme di clausole scritte in Forma Normale Congiunta[1] e definite da una sintassi standard compatibile con frammento CNF senza uguaglianza della libreria *TPTP* (vedi [7]). Si è quindi implementata una procedura di semi-decisione che, basata su un sistema di cinque regole di inferenza con ordinamenti (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 etc. L'ordinamento scelto per il dimostratore è l'ordinamento di Knuth-Bendix ordinamento di semplificazione che gode delle proprietà di monotonia, stabilità e proprietà del sotto-termine. Si è scelto di utilizzare il linguaggio di programmazione Java, vista la già affermata conoscenza individuale sul linguaggio, e viste le strutture dati utili già fornite nell'implementazione base.

2 Scelte progettuali ed implementative

Come già detto, 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. Il progetto è diviso in 3 package principali, che sono:

- `data_structures`;
- `main_package`;
- `parser`;

2.1 parser

Per leggere le formule nel formato di TPTP, è stato creto un parser in grado di leggere i file con estensione "*.p*". Per la generazione del parser ci si è affidati a *JavaCC*[9], uno strumento che permette di scrivere file che hanno estensione `.jj`, che definiscono una grammatica context-free BNF (*Backus–Naur Form*) di tipo *LL(1)*. JavaCC genererà poi tutti i file *.java* da compilare per generare il parser. La peculiarità di JavaCC è quella di generare contemporaneamente analizzatore lessicale e sintattico partendo semplicemente dal file che definisce la grammatica. Come da specifica del progetto, è stata implementata la sola porzione di sintassi che definisce le

[1] *CNF*, Conjunctive Normal Form

formule in forma normale congiunta tralasciando gli altri casi. L'implementazione è contenuta nel file `tptpgrammar.jj` (package `parser`) nel quale è stato inframmezzato il codice necessario per la costruzione degli oggetti e strutture dati su cui operare.

2.2 data_structures

Tutte le classi bean[2] sono contenute nel package `data_structures`. *Java* ha anche permesso un minimo di progettazione orientata agli oggetti, in particolare delle classi che definisco i termini (funzioni, variabili e costanti) e i letterali. Molto rilevante la classe astratta `Term`, che è superclasse di `Function`, `Variable` o `Constant`, poiché è spesso necessario riferirsi alla classe che le rappresenta tutte (come ad esempio quando si specificano gli argomenti di un letterale).

2.3 main_package

Le classi contenute nel package `main_package` sono tutte le classi che si occupano di risolvere il problema dato in entrata. Le classi principali sono quelle che si riferiscono alle regole, all'ordinamento, all'unificazione e al ciclo della clausola data.

2.3.1 ExpansionRules

La classe `ExpansionRules` implementa le due regole di espansione *binaryResolution* e *factorization*. Ognuna delle due regole prende in input rispettivamente una o due clausole. Essendo il sistema di inferenza ordinato, entrambe prima di eseguire ogni operazione estrapoleranno i predicati massimali, in modo da eseguire le operazioni di risoluzione e di fattorizzazione solamente sui predicati massimali. Entrambe le regole restituiranno una `LinkedList<Clause>` di clausole. Entrambi i metodi sono statici, e richiamabili da ogni classe del progetto senza dover instanziare oggetti.

2.3.2 ContractionRules

La classe `ContractionRules` implementa le regole di contrazione *isTautology*, *subsumption* e *clauseSimplification*. Sono state implementate due tipologie di algoritmi di sussunzione, e più precisamente sono stati implementati gli algoritmi presentati nel libro di testo del corso (Chang, Lee)[1] e la funzione di Stillman presentata in [6]. L'algoritmo di Stillman è basato principalmente sulla generazione di sostituzioni in successione con backtracking, mentre l'algortimo di Chang-Lee si basa sulla risoluzione. Per le definizioni di complessità si rimanda a [6].

2.3.3 UnificationRules

La classe `UnificationRules` contiene il metodo `mostGeneralUnifier`, che, date due liste di argomenti, l'unificatore più generale[3] tra le due liste. Viene restituita 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 [10]) che è quadratico nella grandezza delle espressioni che devono essere unificate.

2.3.4 KBOComparator

In questa classe troviamo tutte le direttive per l'ordinamento di Knuth-Bendix. Più precisamente `KBOComparator` è una classe che estende `Comparator`, ed effettua l'ordinamento su predicati. Vengono passati in input due predicati, e ordinati in base a tutte le regole dell'ordinameto. I pesi sono standard, e precisamente sono:

- per letterali, variabili e funzioni: 1;
- per costanti: 2;

[2] così si denotano le classi che logicamente contengono le informazioni da manipolare
[3] *MGU*, Most General Unifier

Inoltre l'ordinamento di Knuth-Bendix ha bisogno di una precedenza, che viene data dall'ordinamento lessicografico delle stringhe che rappresentano i simboli di predicati e funzioni.

3 Ciclo della clausola data e Strategia di ricerca

Per l'implementazione del ciclo della clausola data ci si è affidati a due strategie di ricerca, dette ciclo à la Otter e ciclo à la E. Per la scelta della clausola data sono state implementate diverse strategie. La prima di esse consiste nel selezionare la prima clausola seguendo un ordinamento a lista FIFO o LIFO. Il criterio di selezione dipende quindi dall'ordine con cui le clausole vengono lette dal dimostratore. Ovviamente questa strategia non è ottimale, perchè volendo noi arrivare alla clausola vuota, sarebbe più conveniente partire a selezionare le clausole di dimensione più piccola.

Per fare questo è stata implementata una piccola modalità di assegnazione di pesi alla clausola, che consiste semplicemente nell'assegnare peso unitrio ad ogni simbolo della clausola. Questo ci permette di utilizzare una `PriorityQueue` che permette di mantenere tutte le clausole ordinate dalla più piccola alla più grande secondo la dimensione delle stesse.

L'ultimo aspetto rilevante della strategia di ricerca per la scelta della clausola data consiste nell'opzione di poter inserire un valore, chiamato *Peak Given Ratio*, che ogni `k`[4] iterazioni preleva non la clasuola con peso minore, ma quella che da più tempo è in `to_select`. La scelta della clausola data si trasforma quindi in una soluzione mista, visto che la scelta della clausola è data dal metodo `poll()` della coda che restituisce l'oggetto di dimensione minima, ma ogni `k` esecuzioni di questo metodo cambia comportamento, restituendo l'oggeto più vecchio all'interno della coda.

3.1 Ciclo à la Otter

Il ciclo à la *Otter*, selezionabile tramite il comando `-otter`, lavora su due insiemi di clausole, *to_select* e *selected*. Il ciclo va avanti finchè l'insieme *to_select* non viene svuotato, o si incontra una refutazione. Il ciclo prevede, dopo la generazione dei fattori e dei risolventi tramite le operazioni di espansione, di mantenere il più possibile ridotto entrambi gli insiemi contenenti le clausole, così da eliminare tutte le clausole inutili e ripetute. Dopo la fase di espansione vi sono tutte le operazioni di contrazione, che vengono effettuate in questo ordine:

- eliminazione di tautologie, semplificazione clausale e sussunzione su risultato dell'espansione con la clausola data;
- semplificazione clausale e sussunzione su *to_select*;
- semplificazione clausale e sussunzione su *selected*;

Tutto ciò che sopravvive a questa fase viene insierito nell'insieme delle clausole da selezionare, e il ciclo ricomincia.

3.2 Ciclo à la E

Anche il ciclo à la *E*, selezionabile tramite il comando `-e`, lavora su due insiemi di clausole, *to_select* e *selected*. Anche qui si aspetta di svuotare l'insieme *to_select* per terminare, o quando si incontra una refutazione. La differenza sta nel fatto di essere più conservatore nell' applicare le regole di contrazione. Tali regole (come la sussunzione) sono abbastanza come costo computazionale e applicarle all'unione dei due insiemi da selezionare e selezionate potrebbe risultare inutilmente dispendioso, questo perchè viene permesso solo agli elementi in selezionate di essere "genitori" di nuove clausole. Questo però implica di permettere all'insieme *to_select* di avere clausole ripetute. Le operazioni del ciclo vengono eseguite in ques'ordine:

- semplificazine clausola della clausola data con l'insieme selezionate;

[4] di solito il *Peak Given Ratio* è compreso tra 4 e 6

- fase di espansione;
- eliminazione di tautologie, semplificazione clausale e sussunzione sul risultato dell'espansione;
-

Anche in questo metodo, al termine della contrazione, le clausole sopravvissute vengono aggiunte all'insieme da selezionare e il ciclo ricomincia selezionando una nuova clausola. Il compromesso ottenuto è quello di risparmiare tempo di computazione (vengono fatte meno contrazioni) a discapito dello spazio (l'insieme da selezionare può crescere notevolmente).

4 Parametri di esecuzione

Questa è una raccolta dei parametri con cui si può settare il dimostratore di teoremi.

- `-min`, `-max`, `-fifo`: permette di scegliere in che tipo di struttura dati costruire l'insieme *to_select*. Il default è l'opzione della coda di min.
- `-subs=cl` o `-subs=st`: permette di scegliere che algoritmo di sussunzione utilizzare, e cioè quello offerto da Chang-Lee (opzione *cl*) o quello offerto da Stillman (opzione *st*). Il default è Stillman.
- `-peak=n`: permette di inserire un valore per il *Peak Given Ratio*, inserendo un numero al posto di n. Di default il procedimento del peak given ratio non è attivo.
- `-time=n`: permette di inserire un tempo di timeout in secondi per l'esecuzione del dimostratore. Di default non è impostato nessun timeout.

5 Benchmark

Vediamo ora alcuni benchmark del dimostratore, con alcuni esempi tratti dalla libreria TPTP. Tutte le simulazioni sono state fatte su un notebook Asus N56VZ, con CPU `Intel i7-3630QM 2.4GHz, TurboCore 3.4GHz`, 8 Gb di RAM, sistema operativo `Ubuntu 12.10`. Uno degli argomenti che possono essere inseriti è `-time` seguito da un numero che indica il limite massimo di secondi entro cui il ciclo della clausola data può cercare una prova di insoddisfacibilità o un modello per la soddisfacibilità. Ovviamente nel caso questo parametro non sia specificato il tempo di esecuzione rimante potenzialmente infinito. Inoltre se il numero di clausole generate per regole di espansione siano tali da eccedere l'heap space, allora verrà lanciata un eccezione e il programma terminerà brutalmente.

File	Status	Otter contr	Otter exp	E contr	E exp
ALG002-1	UNSAT	445	2523	4695	3174
ANA002-1	UNSAT	12266	13106	27285	62456
ANA004-5	UNSAT	6410	6410	6828	6410
CAT007-3	UNSAT	245	260	561	260
KRS006-1	SAT	61311	time expired	time expired	time expired
NUM284-1.014	UNSAT	3610	out of memory	11509	out of memory
PUZ001-3	SAT	148	152	305	160
PUZ012-1	UNSAT	234	239	467	250
PUZ018-1	UNSAT	time expired	out of memory	time expired	out of memory
SYN086-1.003	SAT	757	813	8393	813
SYN087-1.003	SAT	2583	2621	66101	2621

Tabella 1: Numero di clausole generate dalle regole di espansione

File	Risultato	Otter contr (ms)	Otter exp (ms)	E contr (ms)	E exp (ms)
ALG002-1	UNSAT	981	2833	3537	2793
ANA002-1	UNSAT	8735	9354	16938	32813
ANA004-5	UNSAT	42079	45167	16853	14107
CAT007-3	UNSAT	141	216	278	230
KRS006-1	SAT	123598	time expired	time expired	time expired
NUM284-1.014	UNSAT	30398	out of memory	33460	out of memory
PUZ001-3	SAT	120	146	199	159
PUZ012-1	UNSAT	196	325	606	351
PUZ018-1	-	time expired	out of memory	time expired	out of memory
SYN086-1.003	SAT	365	393	1477	439
SYN087-1.003	SAT	1392	2489	7287	2594

Tabella 2: Tempistiche a seconda della strategia

5.1 Considerazioni

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), Towards Efficient Subsumption, *Lecture Notes in Computer Science. Springer Verlag*.

[4] Löchned B. (2006), Things to Know When implementing KBO, *Springer Science*.

[5] Schulz S. (1999), Simple and Efficient Clause Subsumption with Feature Vector Indexing, *Springer Science*.

[6] Gottlob G. Leitsch A. (1985), On the Efficiency of Subsumption Algorithms, *Journal of the ACM*.

Siti consultati

[7] TPTP, `http://www.cs.miami.edu/~tptp/`

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

[9] T. Norvell - JavaCC tutorial, `http://www.engr.mun.ca/theo/JavaCC-Tutorial/javacc-tutorial.pdf`

[10] AIMA Algorithms, `http://aima.cs.berkeley.edu/algorithms.pdf`