Il metodo dei Tableaux in Prolog

1 Scopo del progetto

Lo scopo del progetto è quello di creare un dimostratore automatico per formule ben formate della logica del primo ordine usando il metodo dei Tableaux.

1.1 Il metodo dei Tableaux

Il metodo dei tableaux è un procedura di dimostrazione estremamente elegante ed efficiente per la logica proposizionale e predicativa. Il "Teorema di Completezza per i Tableaux" (pag.28 Smullyan) ci assicura che :

(a) se la nostra formula ben formata X è una tautologia, allora ogni tableaux completo partendo dalla sua formula negata (FX) deve chiudere.

(b) Ogni tautologia è dimostrabile con il metodo dei tableaux.

Quindi il nostro dimostratore è stato pensato allo stesso modo, cioè quando l'utente chiede al programma *? dimostra(X)*, dove X rappresenta una qualunque formula ben formata della logica predicativa, questa viene negata ($\sim X$)ed iniziano ad essere svolti (eliminati) tutti i connettivi e quantificatori fino a che non rimangono solo le formule atomiche. Se tutti i rami del tableaux chiudono, allora la negata della formula di partenza è insoddisfacibile e quindi la formula è valida.

1.2 Realizzazione

breve preambolo
In principio la nostra difficoltà consisteva nel dover rappresentare il tableaux, quando invece avremmmo potuto usufruire più saggiamente delle caratteristiche peculiari del Prolog. Rinunciare a rappresentare il tableaux e sfruttare il back-tracking del Prolog si è dimostrata una via più fruttuosa di quella che, con perniciosa ostinatezza, volevamo intraprendere.
fine del breve preambolo

Per prima cosa abbiamo definito i connettivi principali :
$\rightarrow$ implicazione
$\vee$ disgiunzione inclusiva (vel)
$\wedge$ congiunzione
$\sim$ negazione
usando il predicato di sistema ternario *op(A,B,C)/3*, dove il primo argomento indica la priorità, ovvero quale connettivo verrà svolto per prima, il secondo argomento l'arità del connettivo ed il terzo il simbolo adoperato per rappresentarlo. Quindi abbiamo scritto

```
:- op(903,xfy,->).
:- op(902,xfy,+).
:- op(901,xfy,^).
:- op(900,fx,~).
```

Poi abbiamo definito il predicato unario *dimostra(X)/1* (che accetta la formula ben formata che si desidera dimostrare) attraverso il predicato ternario *tabl(Lista delle Formule da Sviluppare, Lista delle Formule non più riducibili, Numero di parametri usati)*, che come primo argomento è una lista contenente le formule da sviluppare, il secondo è una lista per le formule dove non sono presenti connettivi: ad esempio formule ground. Il terzo argomento è il numero di parametri istanziati fino a quel punto (del calcolo). Come abbiamo già spiegato la dimostrazione avviene negando la formula di partenza e svolgendo uno alla volta tutti i connettivi, quindi la formula presente in *dimostra(X)/1* viene negata e posta nel primo argomento di *tabl/3*.
All'inizio ovviamente il secondo argomento è una lista vuota ed il numero di parametri usati è zero. Quindi abbiamo scritto

```
dimostra(Formula) :- tabl([~Formula],[],0).
```

Passiamo quindi a definire *tabl/3*. Dichiarativamente il predicato *tabl* è vero se il ramo del tableaux è privo di formule da sviluppare e chiuso. Vedremo in seguito che significa *chiuso*.
Se le due liste son vuote, indipendentemente dal numero di parametri, *tabl* ha comunque successo perchè una formula vuota è sempre valida (chi non dice niente, ha sempre ragione). Grazie all'unificazione ogni clausola di *tabl/3* controlla il tipo di formula, questa potrà essere di tipo congiuntivo, disgiuntivo o con quantificatori.
Iniziamo con le formule di tipo congiuntivo, queste possono essere della forma : $\sim\sim$X, X$\wedge$Y, $\sim$(X$\vee$Y), $\sim$(X$\rightarrow$Y) dove X ed Y sono formule ben formate complicate quanto si vuole (con una loro struttura interna).
In questo caso la formula verrà svolta e le formule restanti X e Y saranno poste in fondo alla lista delle formule da sviluppare grazie al predicato

ternario *aggiungi_in_fondo([X,Y],Xs,Ys)* che attacca, tramite *append*, ordinatamente, la lista Xs ad [X,Y] e la pone in Ys, quest'ultima sarà il primo argomento di *tabl* poichè X ed Y devono essere ulteriormente sviluppate. Allora in questo caso abbiamo scritto

```
tabl([],[],N).
tabl([~(~X)|Xs],A,N) :- tabl([X|Xs],A,N).
tabl([X^Y|Xs],A,N) :-
aggiungi_in_fondo([X,Y],Xs,Ys),tabl(Ys,A,N).
tabl([~(X+Y)|Xs],A,N) :-
aggiungi_in_fondo([~X,~Y],Xs,Ys),tabl(Ys,A,N).
tabl([~(X->Y)|Xs],A,N) :-
aggiungi_in_fondo([X,~Y],Xs,Ys),tabl(Ys,A,N).
aggiungi_in_fondo(Lista1,Lista,Ris) :-
append(Lista,Lista1,Ris).
```

Le formule di tipo disgiuntivo sono della forma : X$\vee$Y, $\sim$(X$\wedge$Y), X$\rightarrow$Y. In questo caso il ramo si divide in due, e sarà chiuso solo se entrambe le strade chiudono. Dichiarativamente si scrive:

```
tabl([X+Y|Xs],A,N) :- tabl([X|Xs],A,N),tabl([Y|Xs],A,N).
tabl([~(X^Y)|Xs],A,N) :- tabl([~X|Xs],A,N),tabl([~Y|Xs],A,N).
tabl([X->Y|Xs],A,N) :- tabl([Y|Xs],A,N),tabl([~X|Xs],A,N).
```

Oppure le formula possono contenere dei quantificatori. A questo scopo abbiamo introdotto due predicati binari :

perogni(X,P) che traduce la formula $\forall xP$

esiste(X,P) che traduce la formula $\exists xP$

dove P è una formula ben formata contenete x

Quindi come primo argomento hanno la variabile da quantificare e come secondo hanno la formula contenente la variabile quantificata.
Prima ci siamo occupati delle formule di tipo δ usando la nota relazione : $\sim \forall xP = \exists x \sim P$.
Quindi tradotto in Prolog

```
tabl([~(perogni(X,P))|Xs],A,N) :- tabl([esiste(X,~(P))|Xs],A,N).
```

Di fronte alla formula $\exists xP$ dobbiamo sostituire tutte le occorrenze di x nella formula P con un nuovo parametro che non sia uno di quelli usati fino a quel

punto del calcolo, per questo motivo scriviamo N1 is N+1 e introduciamo il predicato quaternario *sost(X,P,P1,N1)* che sostituisce con N1 tutte le occorrenze di X in P e crea la nuova formula P1. Le specifiche di *sost* verranno illustrate in seguito. Quindi

```
tabl([esiste(X,P)|Xs],A,N) :- N1 is (N+1),sost(X,P,P1,N1),
tabl([P1|Xs],A,N1).
```

Passiamo ora alle formule di tipo γ usando la nota relazione :
$\sim \exists x P = \forall x \sim P$
quindi in Prolog

```
tabl([~(esiste(X,P))|Xs],A,N) :- tabl([perogni(X,~(P))|Xs],A,N).
```

Il caso dell'istanziazione di una formula γ è più complicato. Poichè questo tipo di formule non si *esauriscono* mai. La via più conveniente, per cercare una contraddizione, è istanziare la formula γ su tutti gli M parametri usati fino a quel momento. Bisogna però avere la formula *pronta* perchè si potrebbe raggiungere una contraddizione per qualche parametro maggiore di M. Quindi la aggiungiamo alla lista delle formule già sviluppate di *tabl*.
Per istanziare una formula γ su più parametri abbiamo dichiarato un nuovo predicato *sostgamma(X,P,I,N,L)/5* che sostitisce tutte le X in P con tutti i parametri da I ad N (con $N > I$) riempendo la lista L con queste nuove formule istanziate.
Vedremo in seguito come agisce nel dettaglio *sostgamma(X,P,I,N,L)*.
Quando la formula γ viene sviluppata per la prima volta si istanzia da zero ad N, poi usiamo il predicato *append(L,Xs,R)* per attaccare la lista L ad Xs formando la nuova lista R, infine si passa a sviluppare il ramo con le formule R.

```
tabl([perogni(X,P)|Xs],A,N) :- sostgamma(X,P,0,N,L),
append(L,Xs,R),tabl(R,[perogni(X,P,N)|A],N).
```

Ricordando che il Prolog applica le clausole nell'ordine in cui sono scritte all'interno del programma l'ultima clausola viene applicata solo quando l'unificazione con le precedenti fallisce: significa in questo caso che la formula in cima alla lista non è ulteriormente sviluppabile, in questo caso si aggiunge alla lista A e si passa a sviluppare le rimanenti Xs.

```
tabl([X|Xs],A,N) :- tabl(Xs,[X|A],N).
```

Se tutte le formule da sviluppare sono terminate (la lista si è svuotata), il programma va a vedere se la lista A chiude. Per far ciò abbiamo deciso di dichiarare il predicato unario *chiude(A)* che ha successo se trova nella lista

in questione una contraddizione, cioè se trova due elementi in A del tipo X e ∼X. Se ciò si verifica allora appare a video la scritta "formula valida" ed il predicato tabl è vero (tabl è vero se il ramo associato è stato sviluppato e chiude). Altrimenti se questo goal fallisce ed il numero di parametri N è maggiore di 10 appare la scritta "non lo so".

```
tabl([],A,N) :- chiude(A),write('formula valida'),!.

tabl([],A,N) :- N>10,write('non lo so'),nl,!.

chiude(A) :- member(X,A),member(~X,A).
```

Nel caso in cui dovessero fallire i primi due goal precedenti, cioè la lista non chiude ed $N < 10$, si incrementa N di una unità (N1 is N+1) e si controlla con il predicato *member/2* la presenza di eventuali formule γ nella lista A delle *formule non più riducibili* di *tabl* che possono essere ancora usate al fine di far chiudere il ramo. Se *member* la trova (ha successo) esce a video la scritta : *provo a reistanziare la formula perogni(X,P).* Fino a questo momento la formula P è stata istanziata su M parametri. Si procede quindi ad istanziare la formula P da M ad N con *sostgamma* nel tentativo di far chiudee il ramo. Il programma ci prova fino a $N = 10$. Se il ramo non chiude tutte le volte ed $N > 10$ allora appare a video la scritta *non lo so.* Se invece *member/2* fallisce, ovvero non ci sono più formule γ in A, il ramo non chiude e ne viene mostrato il controesempio.

```
tabl([],A,N) :- N1 is (N+1),
        member(perogni(X,P,M),A),
        write('provo a reistanziare la formula   '),
        write(perogni(X,P)),nl,
        sostgamma(X,P,M,N1,L),
        rimuovi(perogni(X,P,M),A,Ris),
        aggiungi_in_fondo([perogni(X,P,N1)],Ris,B)
        tabl(L,B,N1).
tabl([],A,N) :- write('non valida'),nl,write(A),nl,write('ecco un
controesempio.'),nl,fail.
rimuovi(E,[E|List],List).
rimuovi(E,[X|List],[X|Ris]) :- E\=X,rimuovi(E,List,Ris).
```

Non abbiamo però ancora visto come funzionano *sost/4* e *sostgamma/5.* Vediamo prima come agisce *sost(X,P,P1,N).* Dichiarativamente sost/4 è vero se P1 è la formula P dove le occorenze di X sono state sostituite da N. La definizione di sost/4 deve essere ricorsiva per entrare nella struttura di P. I

casi base della ricorsione sono: $P = X$ e"P è un atomo diverso da X". Nel primo caso viene sostituito P con N, nel secondo P resta invariato. L'ultima clausola gestisce invece i termini composti. Questa clausola ricorsiva esplora la formula P con l'aiuto del predicato di sistema *functor(P,Nome,Ari)* che è vero se P è un funtore di nome *Nome* ed arità *Ari*. In questo caso bisogna sostituire X in tutti gli argomenti di P chiamando il predicato ausiliario *sost/5* che esegue iterativamente la sostituzione all'interno dei sottotermini. L'arità del funtore principale del termine (Ari) viene usata come valore iniziale di un contatore di ciclo (A) che viene decrementato in successione (A is A-1) per controllare l'iterazione. Il ciclo ovviamente termina quando $A1 = 0$.

```
sost(X,P,N,N) :- P=X. sost(X,P,P,N) :- atom(P),!,P\=X.
sost(X,P,P1,N) :- functor(P,Nome,Ari),
functor(P1,Nome,Ari),sost(Ari,X,P,P1,N).

sost(A,X,P,P1,N):- A>0,
                        arg(A,P,Arg),
                        sost(X,Arg,SostArg,N),
                        arg(A,P1,SostArg),
                        A1 is A-1,
                        sost(A1,X,P,P1,N).
sost(0,X,P,P1,N).
```

Passiamo ora a descrivere *sostgamma/5*. Questo predicato provvede alla sostituzione di X nelle formule γ con i parametri da N ad I con $N > I$ (decrementando N). La prima clausola provvede a terminare il ciclo quando il valore di N raggiunge quello di I. Durante il ciclo *sostgamma/5* riempe la lista L con le formule P1 in cui è avvenuta la sostituzione effettuata da *sost/4*

```
sostgamma(X,P,I,I,[]).
sostgamma(X,P,I,N,[P1|L]) :- N>I,
            sost(X,P,P1,N),
            N1 is (N-1),
            sostgamma(X,P,I,N1,L).
```

2 Utilizzo del programma

Usando un interprete prolog (Il programma è stato testato sullo swi-prolog e sul gnu-prolog) si consulta il programma, il predicato *dimostra* traduce la richiesta di dimostrare una formula ben formata della logica del primo ordine.

l'utilizzo è *dimostra(formula)* dove *formula* è una formula ben formata dove i connettivi logici vanno espressi con gli operatori prolog definiti all'inizio, e cioè:

$\sim$ prefisso per la negazione.

$\wedge$ infisso per la congiunzione.

$+$ infisso per la disgiunzione.

$->$ infisso per l'implicazione.

Per esempio: chiedere al programma di dimostrare la formula $A \vee \sim A$ bisogna eseguire la seguente richiesta:

```
dimostra(a + ~a).
```

L'output del programma sarà:

```
formula valida

Yes
```