1 Regole di inferenza

La dimostrazione di una formula α é l'applicazione di una sequenza di regole di inferenza alle formule della KB al fine di derivare α.

Regole di inferenza per la logica del primo ordine:

- Eliminazione dell'implicazione (modus ponens): $\frac{a \rightarrow b,\ a}{b}$;
- Eliminazione degli and: $\frac{a_1 \wedge a_2 \wedge \cdots \wedge a_n}{a_i}$;
- Introduzione di and: $\frac{a_1, a_2, \ldots, a_n}{a_1 \wedge a_2 \wedge \cdots \wedge a_n}$;
- Introduzione di or: $\frac{a_i}{a_1 \vee a_2 \vee \cdots \vee a_n}$;
- Risoluzione unitaria: $\frac{a \vee b,\ \neg b}{a}$;
- Risoluzione: $\frac{a \vee b,\ \neg b \vee c}{a \vee c}$;
- Eliminazione del $\forall$: $\frac{\forall v\ \ a}{Sost(\{v/g\}, a)}$ dove g é un termine ground;
- Eliminazione dell'$\exists$: $\frac{\exists v\ \ a}{Sost(\{v/g\}, a)}$ dove g é un termine ground nuovo;
- Introduzione dell'esiste: $\frac{a}{\exists v\ \ Sost(\{g/v\}, a)}$ dove g é un termine ground che compare in a
- Modus Ponens generalizzato: $\frac{P'_1, \ldots, P'_n,\ (P_1, \ldots, P_n) \rightarrow Q}{Sost(\theta, Q)}$

  se $\exists \theta$ tale che $Sost(\theta, P'_i) = Sost(\theta, P_i)\ \ \forall i \in \{1, \ldots, n\}$.
- Risoluzione generalizzata: $\frac{P_1 \vee \cdots \vee P_j \vee \cdots \vee P_m,\ Q_1 \vee \cdots \vee Q_k \vee \cdots \vee Q_n}{Sost(\theta, P_1 \vee \cdots \vee P_{j-1} \vee P_{j+1} \vee \cdots \vee P_m \vee Q_1 \vee \cdots \vee Q_{k-1} \vee Q_{k+1} \vee Q_n)}$ dove $\theta = Unify(P_j, \neg Q_k)$.

La base di conoscenza é espressa in CNF (congiunzione di disgiunzioni).

Comunque essendo la logica del primo ordine semidecidibile se la formula da dimostrare non é implicata dalla base di conoscenza la procedura di dimostrazione puó non terminare.

La refutazione (o dimostrazione per assurdo) mediante risoluzione è una procedura di inferenza completa. L'idea è quella che per dimostrare P si assume P falso. Se la dimostrazione porta a una contraddizione la base di conoscenza implica P. In formule: $((KB \wedge \neg P) \models Falso) \Leftrightarrow (KB \models P)$.

2 Dimostratore di Teoremi

I dimostratori di teoremi usano regole di inferenza per dimostrare formule in logica del primo ordine. Distiguiamo tre tipi di strategie di risoluzione:

- **Preferenza dell'unità**: Si applica la risoluzione, dove una delle formule è costituita da un singolo letterale (clausola unitaria). L'idea è quella di produrre formule via via più corte. Infatti:
  P (formula unitaria)
  $P \wedge Q \rightarrow R$
  $Q \rightarrow R$ (formula più breve).

- **Risoluzione di Input**: Si applica la risoluzione tra una delle formule di input (KB+query) e un'altra formula.
  NOTA: La risoluzione di input è completa per basi di conoscenza in forma di Horn ($P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q$).

- **Insieme di Supporto**: Si definisce un sottoinsieme delle formule, chiamato "insieme di supporto". Si applica la risoluzione tra una delle formule dell'insieme di supporto e un'altra formula; il risultato dell'inferenza viene aggiunto all'insieme di supporto.
  NOTA: Una scelta errata dell'insieme di supporto rende la risoluzione basata su insieme di supporto incompleta. Se si sceglie l'insieme di supporto in modo tale che tutte le rimanenti formule siano tutte contemporaneamente soddisfacibili, la risoluzione basata su insieme di supporto è completa (un modo è quello di mettere l'interrogazione negata nel sos).

2.1 Applicazioni dei Dimostratori di Teoremi

- **Dimostrazione di teoremi matematici**.
  SAM (Semi-Automated Mathematics) è stato il primo a dimostrare un lemma delle teoria dei reticoli.
  Il dimostratore di teoremi di Boyer-Moore ha dato la prima dimostrazione formale del teorema di incompletezza di Godel.

- **Aiuto nelle dimostrazioni**.

  - fornire consigli ai matematici (il matematico funge da supervisore decidendo la strategia da utilizzare e domandando al dimostratore di teoremi di colmare i dettagli mancanti);
  - verificare dimostrazioni fornite da uomini.

- **Verifica di Software**.
  Un algoritmo viene verificato mostrando che le uscite rispettano le specifiche per ogni possibile ingresso. La verifica di algoritmi è naturalmente molto costosa; per questo motivo è attuata molto raramente. Tuttavia per particolari applicazioni quali ad esempio il programma per il controllo delle fissioni all'interno del nocciolo di una centrale nucleare o particolari

transizioni bancarie, lo sforzo di utilizzare dimostratore di teoremi è giustificabile.
L'algoritmo di criptazione a chiave pubblica RSA e l'algoritmo di comparazione tra stringhe di Boyer e Moore sono stati verificati da un dimostratore di teroremi.

- **Verifica di Hardware**.
  Un circuito viene verificato mostrando che le uscite rispettano le specifiche per ogni possibile ingresso. Le formule descriveranno le interazioni fra i segnali e gli elementi del circuito.
  Dimostratori di teoremi progettati appositamente per la verifica sono stati in grado di verificare intere CPU.

- **Sintesi di algoritmi software**.
  L'idea é quella di dimostrare un teorema del tipo: "esiste un programma P che soddisfi una certa specifica". La sintesi di algoritmi guidata da utente è stata utile nella progettazione di algoritmi nuovi e sofisticati.

- **Sintesi di hardware**.
  AURA è stato il primo dimostratore di teoremi ad essere utilizzato per il progetto di circuiti più compatti.

3 Otter

3.1 Premessa

Otter è un dimostratore di teoremi, dichiarati in logica del primo ordine con l'operatore di uguaglianza.

Otter utilizza le seguenti regole di inferenza:

- Binary resolution (flag `bin_res`): date due clusole in conflitto permette di derivate la clausola vuota. Ad esempio:

  $$\frac{nurse(Roberta), \neg nurse(Roberta)}{False};$$

- Unit resulting resolution (flag `ur_res`): Il risultato deve essere una clausola unitaria. Data una clausola con n termini e $n-1$ clausole unitarie le risolve in un colpo solo.
  Ad esempio:

  $$\frac{\neg P(x,y) \vee Q(y,z) \vee \neg R(u,v) \vee S(x,z), \;\; P(John,Mary) \wedge \neg Q(Mary,Cats) \wedge R(Birds,Mice)}{S(John,Cats)};$$

- Hyper-Resolution (flag `hyper_res`): L'hyper resolution coinvolge più clausole. Una clausola deve avere almeno un letterale negativo, mentre le altre clausole non devono averne. Risolvendo i letterali negativi con le altre clausole positive, la risultante clausole non conterrà letterali negativi. Ad esempio:

  $$\frac{\neg P(x,y) \vee Q(x) \vee \neg R(y) \;\; P(Spock,Data) \;\; R(z) \vee S(z)}{Q(Spock) \vee S(Data)};$$

- Paramodulation: La paramodulazione considera due clausole genitori: from_clause e into_clause. Uno degli argomenti del predicato di uguaglianza deve unificare con un sottotermine dell'altra clausola. Dopo l'unificazione dell'argomento, il sottotermine è rimpiazzato dall'argomento del predicato di uguaglianza.
  Ad esempio:

  $$\frac{difference(two_sqaured, b) \;\; equal(two_squared, four)}{difference(four, b)}.$$

  In particolare distinguiamo due tipi di paramodulazione:

  - Paramodulation from the given clause (flag `para_from`): la given clause contiene un predicato di uguaglianza;
  - Paramodulation into the given clause (flag `para_into`): il predicato di uguaglianza non è contenuto nella given clause, ma il risultato dell'inferenza è la sostituzione di un parametro della given_clause.

Queste regole di inferenze considerano un piccolo insieme di clausole e ne inferiscono una. Se quest'ultima è interessante viene memorizzata e può essere utilizzata per successive inferenze.

Altre caratteristiche di Otter sono:

- le asserzioni possono essere inserite come formule del primo ordine o come clausole (disgiunzione di letterali con quantificatori universali impliciti). Otter trasforma le formule in logica del primo ordine in clausole;
- Forward demodulation: semplifica le nuove clausole inferite mediante un insieme di uguaglianze;
- Backward demodulation: semplifica le clausole esistenti utilizzando le nuove uguaglianze inferite (quelle aggiunte all'insieme dei demodulatori);
- Forward sussumption: cancella la clausola inferita se essa è sussunta da una clausola esistente;
- Backward sussumption: cancella le clausole esistenti sussunte dalla clausola inferita;
- una variante del metodo di Knuth Bendix (il metodi di Knuth Bendix tenta di trasformare un insieme di uguaglianze in un insieme di demodulatori);
- pesi e ordinamento lessicale guidano la scelta della clausola migliore (le clausole migliori sono quelle di peso inferiore che corrispondono all'incirca a quelle formate da un numero minore di disgiunti).

L'utente codifica un problema in logica del primo ordine o in clausole, sceglie le regole di inferenza, setta le opzioni per controllare il processo di inferenza e decide quali formule o clausole appartengono all'insieme di supporto e quali uguaglianze (se presenti) appartengono all'insieme dei demodulatori.

4 L'algoritmo di Otter

Otter lavora solo ed esclusivamente con clausole. Se l'utente inserisce delle formule in logica del primo ordine, Otter le trasforma in clausole; se ciò non è possibile Otter termina immediatamente. Otter gestisce quattro liste di clausole:

- usable: questa lista contiene le clausole usate per l'inferenza;
- sos: contiene le clausole che partecipano alla ricerca (ad ogni passo dell'algoritmo si sceglie la given_clausola da sos);
- passive: non partecipano alla ricerca, sono usate sole per sussunzione in avanti e unit conflict. L'algoritmo di sussunzione in avanti cancella la nuova clausola inferita se è sussunta da quelle presenti in "usable", "sos" e "passive". Le clausole unitarie inferite vengono controllate rispetto a quelle presenti in "usable", "sos" e "passive" alla ricerca di eventuali unit-conflict;[1]
- demodulators: sono un insieme di uguaglianze utilizzate per riscrivere le nuove clausole inferite.

Principalmente l'algoritmo di Otter utilizza le liste usable e sos.

```
while (sos vuota & nessuna dimostrazione e' stata trovata)
begin
  given_clause:=migliore_clausola(sos);
  usable:=usable ∪ given_clause;
  sos:=sos - given_clause;
  new_clause=inferisci(given_clause, clause_usable)
  if retention_test(new_clause) then
     sos:=sos ∪ new_clause;
endwhile.
```

La strategia del sos (utilizzata da otter) richiede di dividere le clausole in due insiemi, quelle con supporto (sos) e quelle senza (usable). All'inizio le sos sono le clausole supportate e le usable sono quelle non supportate. Ma dopo che l'algoritmo è iniziato le clausole in usable non corrispondono più alle clausole non supportate in quanto in essa finiscono le given_clauses (sos).

Ad esempio:

```
set(binary_res).

list(usable).
-citta(x) | -citta(y) | -aereo(z) | -at(z,x) | at(z,y).
```

[1] La lista passiva è utile per controllare l'avanzamento del processo di ricerca in dimostrazioni difficili. Supponiamo di avere un difficile teorema e dei lemmi inerenti. Inserendo i lemmi negati nella passive list il processo di ricerca non verrà influenzato e potremmo sapere se otter riesce a dimostrarli.

```
citta(Brescia).
citta(Milano).
aereo(a1).
-at(a1,Milano).
end_of_list.

list(sos).
at(a1,Brescia).
end_of_list.
```

Qual'è il vantaggio di utilizzare una clausola in sos e tutte le altre in usable? Per limitare la ridondanza.
Ad es.
A & B $\rightarrow$ C in usable;
A e B in sos;
se non si seguisse la restrizione potrei derivare C due volte: Supponiamo di prendere come given_clause = A; con la clausola B si potrebbe dedurre C e poi mettere A in usable. Nel successivo ciclo dell'algoritmo si potrebbe prendere come given_clause B e tramite A dedurre nuovamente C

5 Avvio

Per utilizzare Otter eseguire:
otter < file-problema > file-soluzione.

6 Sintassi

Otter riconosce clausole e formule. Le clausole sono disgiunzioni di predicati le cui variabili sono quantificate universalmente in modo implicito. Le formule del primo ordine necessitano di una quantificazione esplicita.

Otter distingue le stringhe in tre categorie:

- nomi ordinari: una stringa composta da caratteri alfanumerici,$,_.
- nomi speciali: una stringa composta da caratteri nell'insieme: $*+-/\^{<}>=':$?@&!; # e qualche volta (vedi sotto) |.
- nome quotato: qualsiasi stringa racchiusa tra "" o ''.

I restanti simboli stampabili sono:

- %: commento
- .: termina espressione;
- ,()[] e qualche volta (vedi sopra) |: punteggiatura e simboli di raggruppamento.

Tabella 1: Lista di simboli logici per la costruzione di formule.

Operazione	Simbolo	Priorità
Negazione	-	1
Congiunzione	&	2
Disgiunzione	—	3
Implicazione	$\rightarrow$	4
Equivalenza	$\leftrightarrow$	4
Quantificatore esistenziale	exists	-
Quantificatore universale	all	-

Determinare se un termine é una costante o una variabile dipende dal contesto. All'interno di una clausola (quantificatore universale implicito) il simbolo determina il tipo (variabile, costante). La regola di default dice che una variabile inizia per u, v, w, x, y, z. Se il flag prolog_style_variables é settato una variabile inizia unicamente con _ o una lettera maiuscola.

All'interno di una formula un simbolo é una variabile se e solo se é quantificato.

I nomi che iniziano con $ sono riservati. Il nome = e tutti i nomi che iniziano con eq, Eq, EQ sono riconosciuti come predicati di uguaglianza dai processi di paramodulazione e di demodulazione.

6.1 Termini e Atomi

I termini sono valutati come oggetti, gli atomi come valori di verità. Costanti e variabili sono termini. Un simbolo di funzione n-aria applicata ad n termini é anch'essa un termine. Un simbolo di predicato n-ario applicato ad n termini é un atomo.

Come scrivere termini e atomi?
Un simbolo di funzione o un simbolo di predicato + "(" + argomenti separati da virgole + ")"

Gli spazi bianchi possono apparire dovunque tranne che tra il simbolo di funzione e l'apertura della parentesi.

6.2 Clausole Letterali

Un letterale é un atomo o una negazione di un atomo.
Una clausola é una disgiunzione di letterali. Le clausole possono essere scritte in forma prefissa o infissa.

6.3 Formule

Una formula é un insieme di letterali combinati usando i simboli in tabella **??**.
Le formule possono essere scritte in forma prefissa o infissa.
Per la quantificazione di una formula in forma prefissa si usa il funtore $Quantified.
Ad esempio $Quantified(all, x, y, exists, z, |(P(x,y), -(Q(z)))). In forma infissa

Tabella 2: Comandi.

Comando	Descrizione
include(file_name)	include file_name nel file corrente
op(precedenza,tipo,nome)	dichiara l'operatore nome
make_evaluable(simbolo,simbolo_valutato)	dichiara un alias di simbolo_valutato
set(flag_name)	setta il flag flag_name
clear(flag_name)	resetta il flag flag_name
assign(parameter_name,integer)	parameter_name=integer
list(list_name)	lista delle clausole
formula_list(list_name)	lista delle formule
weight_list(weight_list_name)	lista dei pesi
lex(symbol_list)	assegna un ordinamento ai simboli
skolem(symbol_list)	identifica le funzioni di skolem
lrpo_multiset_status(symbol_list)	stato di lrpo

l'esempio diviene: "all x all y exists z (P(x,y)|-Q(z))" la quantificazione universale puó essere semplificata in "all x y".
Gli spazi bianchi sono richiesti prima e dopo "all" ed "exists" e prima di "-".

7 Comandi ed il File di Input

L'input di Otter consiste in un insieme di comandi alcuni dei quali indicano delle liste. Tutte le liste terminano con end_of_list. I comandi sono elencati nella tabella **??**.

Otter riconosce due tipi di opzioni: flag e parametri.
I flag sono opzioni booleane che vengono cambiate con set e clear.
es.: set(binary_res).

I parametri sono valori interi e sono cambiati con il comando assign.
es.: assign(max_seconds,300).

Le liste possono essere dei seguenti tipi:

- usable (clausole e formule)
- sos (clausole e formule)
- demodulators (clausole)
- passive (clausole e formule)

es.: list(usable), formula_list(sos).
Se il file di input contiene liste dello stesso tipo sono concatenate.

7.1 La Modalià' Automatica

La modalitá automatica é impostata tramite il flag auto (set(auto)). Il flag deve essere settato come primo comando e tutte le clausole devono essere messe nella

lista usable. In questa modalitá Otter esamina le clausole di input per decidere le regole di inferenza e la strategia di ricerca. Otter sposterá alcune clausole da usable a sos prima che inizi la ricerca.