ALCUNE CONSIDERAZIONI SULLA INTEGRAZIONE TRA LIVELLO OGGETTO E META NEL LINGUAGGIO LOGICO MI_PROLOG.

L.Giordano, A.Martelli, I.Murgia, G.F.Rossi.

Dipartimento di Informatica
C.so Svizzera 185, 10149 Torino, Italy.

RIASSUNTO

In questo lavoro viene presentata una estensione del Prolog, MI_Prolog, definita secondo l'approccio di Bowen e Kowalski, che consiste nell'amalgamare linguaggio oggetto e metalinguaggio. Di tale approccio viene provata la correttezza nella logica del I ordine. L'estensione e' definita introducendo alcuni metapredicati, mediante i quali le teorie del linguaggio vengono trattate come oggetti del I ordine. Viene fornita una descrizione accurata della sintassi e della semantica di questi metapredicati. In particolare in MI_Prolog viene fornita una soluzione al problema della quantificazione delle variabili all'interno delle primitive add_to e drop_from, introducendo una distinzione sintattica tra variabili universali ed esistenziali e fornendo un supporto alla loro gestione all'interno del metainterprete. Infine, viene mostrato che le capacita' "meta" di MI_Prolog, ed in particolare la possibilita' di trattare variabili quantificate esistenzialmente, consentono di fare Ragionamento Ipotetico in modo semplice.

INTRODUZIONE.

Il diffuso interesse suscitato nell'ambito dell'AI dalla nozione di "meta_conoscenza" ha prodotto nell'ultimo decennio una serie di sistemi, di ambienti di programmazione e di linguaggi, in cui essa puo' essere rappresentata in modo adeguato.
Una proposta particolarmente interessante e' quella di Bowen e Kowalski, in cui viene affrontato il problema di amalgamare linguaggio oggetto e metalinguaggio, relativamente ai linguaggi logici.
In tale ambito si inquadra la definizione del linguaggio logico MI_Prolog, che consiste in una estensione del Prolog ottenuta con l'introduzione di un metalivello. In MI_Prolog sono stati affrontati e risolti alcuni dei problemi lasciati aperti in [BoKo82, BWei85, BoWe85].
Le caratteristiche principali del linguaggio MI_Prolog sono le seguenti:

- -strutturazione del DB
- -semantica dichiarativa
- -possibilita' di fare Ragionamento Ipotetico

Nel seguito viene presentata brevemente la proposta di Bowen e Kowalski (par. I) ed e' provata la correttezza del loro approccio per quanto concerne i linguaggi del I ordine (par. II). Viene, quindi, definito il linguaggio MI_Prolog (par.IV). Infine, e' affrontato il problema

del Ragionamento Ipotetico in MI_Prolog (par.V) e vengono fatti alcuni cenni all'implementazione di MI_Prolog (par.VI).

I. LINGUAGGIO OGGETTO E METALINGUAGGIO.

Questo paragrafo riassume la proposta di Bowen e Kowalski il cui approccio e' stato seguito per la definizione di MI_Prolog.
Come **linguaggio oggetto** L consideriamo un linguaggio logico, in cui e' definita una relazione binaria di derivabilita' fra insiemi di formule di L e altre formule di L.
Un **metalinguaggio** M del linguaggio L e' un linguaggio logico in cui si possono rappresentare mediante termini le espressioni linguistiche di L. Inoltre, deve essere possibile definire in M una rappresentazione della relazione di derivabilita' in L.
La relazione che permette di associare ogni espressione linguistica di L con un termine di M e' detta **relazione di naming**.

Formalmente:
Sia demo un predicato binario di M. Nel contesto dell'insieme Pr di formule di M, demo rappresenta la relazione di derivabilita' in L sse per ogni insieme finito T (teoria) di formule di L e per ogni formula G di L,

(1) $T \vdash_L G$ sse $Pr \vdash_M$ demo("T","G")

dove "T" e "G" sono termini di M che rappresentano le espressioni linguistiche T e G di L, secondo la relazione di naming utilizzata.

Pr e' un insieme di formule di M che contiene la definizione del predicato demo. Il predicato demo simula in M l'interprete del linguaggio L.

Disponendo di un linguaggio L e di un metalinguaggio M per esso, e' possibile, anziche' fare ragionamenti al solo livello oggetto o al solo livello meta, fare delle deduzioni che coinvolgano entrambi i livelli (in altre parole e' possibile amalgamare L ed M).
Per descrivere in modo formale la relazione che lega la deduzione di un goal in L con la deduzione corrispondente in M, come definito in (1), e' necessario introdurre due regole di inferenza. Queste regole, dette **linking rules** (o regole di riflessione), hanno la forma seguente:

(2)
$$\frac{Pr \vdash_M \text{demo("T","G")}}{T \vdash_L G}$$

(3)
$$\frac{T \vdash_L G}{Pr \vdash_M \text{demo("T","G")}}$$

La prima regola dice che il metalinguaggio puo' comunicare le soluzioni dei problemi al linguaggio oggetto, mentre la seconda dice che il linguaggio oggetto puo' comunicare le soluzioni dei problemi al metalinguaggio.

Secondo la definizione data in [BoKo82], il linguaggio che **amalgama** L e M (indicato nel seguito con L_a) e' costituito da L, M e da:
(a) una relazione di naming;
(b) una rappresentazione di $\vdash_L$ per mezzo di un predicato demo, nel contesto dell'insieme di formule Pr di M;
(c) le regole (2) e (3).

La nozione di linguaggio che amalgama un linguaggio oggetto L ed un suo metalinguaggio M e' introdotta allo scopo di descrivere in modo formale la integrazione del livello oggetto e del livello meta.
Va notato che questa nozione di linguaggio amalgamato risulta in realta' piuttosto vaga. Non e' chiaro infatti, dalla definizione, quale e' l'insieme delle formule ben formate di L_a, quali sono gli assiomi e le regole di L_a e quali le frasi derivabili in L_a. Essendo le linking rules regole di L_a, questo deve essere un linguaggio in cui e' possibile rappresentare la relazione di derivabilita' in L e in M, ossia un metalinguaggio per L e per M.
Nel seguito, per semplicita' e chiarezza, si trattera' dell'integrazione di linguaggio e metalinguaggio senza fare uso della nozione di linguaggio amalgamato.

Nel caso in cui sia L=M, ossia L e' metalinguaggio di se stesso, in L deve essere possibile rappresentare mediante termini ogni espressione linguistica di L stesso (tramite una qualche relazione di naming); inoltre L deve contenere un predicato demo che rappresenti la relazione di derivabilita' in L, nel contesto di un insieme Pr di formule di L. In L si possono quindi scrivere frasi che contengono sia espressioni del linguaggio sia espressioni del metalinguaggio ed in particolare frasi autoreferenziali.

La nozione di relazione di naming e' centrale nella definizione del metalinguaggio di un linguaggio L, poiche' concerne il tipo di rappresentazione che viene associata nel metalinguaggio ad ogni espressione del linguaggio oggetto.
La relazione di naming puo' essere definita in molti modi differenti. Ad esempio, disponendo dei simboli dell'aritmetica (s,+,*,0,<), si puo' rappresentare ogni espressione del linguaggio con il numero di Godel corrispondente. Un'altra scelta, forse piu' naturale, e' quella di rappresentare le espressioni del linguaggio oggetto con termini del metalinguaggio strutturalmente simili.
La rappresentazione scelta deve pero' soddisfare i seguenti requisiti:

(1) ad ogni espressione linguistica del linguaggio oggetto puo' essere associata piu' di una rappresentazione nel metalinguaggio, ma ogni rappresentazione nel metalinguaggio corrisponde ad un'unica espressione del linguaggio oggetto;

(2) le rappresentazioni di ogni espressione di L sono effettivamente calcolabili;

(3) dato un termine di M, e' effettivamente decidibile se esso rappresenta una qualche espressione di L e, in caso affermativo, e' effettivamente calcolabile quale espressione esso rappresenta.

In MI_Prolog e' stata fatta la scelta di rappresentare le espressioni del linguaggio mediante termini strutturalmente simili. Per i dettagli si rimanda al paragrafo III.

II. INTEGRARE LINGUAGGIO OGGETTO E METALINGUAGGIO: IL PROBLEMA DELLA CONSISTENZA.

Quando il linguaggio L ed il suo metalinguaggio M sono due linguaggi differenti, entrambi consistenti, la loro integrazione tramite l'uso delle linking rules non comporta problemi di inconsistenza. Infatti, l'introduzione delle linking rules non accresce il potere deduttivo di L o di M; le linking rules sono "sovrabbondanti". Come viene osservato in [BoKo82], pag.164, "per l'assunzione della rappresentabilita', ogni applicazione di una linking rule puo' essere rimpiazzata da una dimostrazione equivalente condotta interamente in L o in M".

Il caso in cui linguaggio e metalinguaggio coincidono (L=M) e' piu' problematico.
Cio' che si intende dimostrare e' che un linguaggio L puo' essere metalinguaggio di se stesso (e quindi essere capace di autodescriversi, cioe' di rappresentare ogni sua espressione mediante un termine) ed essere allo stesso tempo consistente.

Sia L1 un linguaggio del I ordine senza assiomi propri. L1 e' chiaramente consistente. Sia L un linguaggio ottenuto introducendo in L1 la lettera predicativa demo e tutte le nuove costanti e lettere funzionali che servono per potere rappresentare in L ogni espressione linguistica di L stesso (l'insieme delle costanti e delle lettere funzionali da introdurre dipende dalla relazione di naming utilizzata). Si puo' dimostrare che L e' una estensione conservativa di L1 ed e' quindi consistente. L e' dunque un linguaggio consistente capace di autodescriversi.

Sia Pr un insieme di formule di L tale che, per ogni teoria T e formula G di L,

(4) $T \vdash_L G$ sse $Pr \vdash_L demo("T","G")$.

Si osservi che L contiene la lettera predicativa demo, ma le formule di Pr che definiscono il predicato demo non sono assiomi propri di L.
Se tale Pr esiste, L e' metalinguaggio di se stesso e, per quanto detto sopra, e' consistente.

In [Kowa83] viene evidenziata la necessita' di includere nel linguaggio L un predicato di naming (ossia un predicato name(x,y) che associ ad ogni espressione y del linguaggio il termine x che la rappresenta) per poter combinare in una stessa frase il metalivello con il livello oggetto. Tale predicato e' necessario, ad esempio, quando una stessa variabile occorre nell'argomento di un predicato e di un metapredicato in una stessa frase, come nella query:

?-demo("T",["p",X]),name(X,Y),q(Y).

In [NaAb87] viene dimostrato che, se L e' un linguaggio capace di autodescriversi ed in cui e' definito un predicato di naming name(x,y) con il significato di "x e' il termine che rappresenta l'espressione y", e' possibile esprimere in L il paradosso del mentitore e di conseguenza L non e' consistente.
Si puo' dimostrare tuttavia che, se L e' un linguaggio del I ordine, questo non e' piu' necessariamente vero.
Intuitivamente, cio' e' dovuto al fatto che, se L e' del I ordine, non e' possibile disporre in L di un predicato name(x,y) definito come sopra: name(x,y) e' definito solamente quando x e' un termine di L che rappresenta a sua volta un termine (e non una generica espressione) di L, in quanto gli argomenti di un predicato devono essere termini del linguaggio.

Piu' precisamente, si puo' dimostrare che il linguaggio ottenuto introducendo in L un predicato di naming con la limitazione precedente e gli assiomi per esso e' una estensione conservativa di L e quindi e' consistente se L lo e'.

Le obiezioni avanzate in [NaAb87] all'approccio che consiste nell'amalgamare linguaggio oggetto e metalinguaggio, suggerito da Bowen e Kowalski, non sono quindi valide quando si considerano linguaggi del I ordine.

III. IL LINGUAGGIO MI_PROLOG.

Il linguaggio MI_Prolog e' una estensione del linguaggio Prolog, ottenuta mediante la definizione del predicato demo e di altri metapredicati, in modo analogo a quanto proposto in [Bowe85, BWei85].

Siano L e M entrambi il linguaggio Prolog. Formalmente, il linguaggio MI_Prolog (MI) e' una estensione di L definita mediante l'introduzione di un metalivello, nel seguente modo:

per ogni formula α di L

$\vdash_{MI} \alpha$ sse $Pr' \vdash_L \alpha$,

dove Pr' e' una metateoria per L, che contiene, oltre all'insieme Pr di formule che definiscono demo, anche un insieme Mt di formule che definiscono gli altri metapredicati introdotti.

Si puo' dimostrare che l'insieme Pr di formule che definiscono demo e' consistente.

PROPOSIZIONE.

Sia L un linguaggio consistente e sia Pr un insieme di formule di L che soddisfano la condizione (4). Pr e' consistente.

Dimostrazione.

Si supponga per assurdo Pr inconsistente. Allora da Pr puo' essere derivata qualunque formula di L, quindi, per ogni teoria T ed ogni formula G di L, si ha

$Pr \vdash_L demo("T","G")$

e , per la (4),

$T \vdash_L G$.

In particolare, per T={} si ha che per ogni formula G di L

$\vdash_L G$

ossia L e' inconsistente, contrariamente a quanto si e' ipotizzato. Quindi Pr e' un insieme di formule consistenti di L.

□

Per quanto dimostrato sopra, il linguaggio MI_Prolog e' consistente, purche' lo sia l'insieme di formule Mt.

Valgono, inoltre, le seguenti relazioni:

$T \vdash_{MI} G$

sse $Pr' \cup T \vdash_L G$

sse $Pr' \vdash_L demo("Pr' \cup T","G")$

sse $\vdash_{MI} demo("Pr' \cup T","G")$

La presenza di "Pr'" nel primo argomento di demo e' intuitivamente necessaria in quanto G in MI_Prolog puo' contenere dei metapredicati.

Il linguaggio MI_Prolog ha la seguenti caratteristiche che lo differenziano dal Prolog standard:

- permette di **strutturare il DB** in teorie;
- rende indifferente l'ordinamento dei goal nel body delle clausole;
- permette di trattare le variabili esistenziali;
- permette di fare **ragionamento ipotetico**.

I predicati di metalivello introdotti sono:

demo(Theory,Goal,Proof)
add_to(Theory1,List,Clause,Theory2)
drop_from(Theory1,List,Clause,Theory2)
current(T)

Essi hanno come argomenti **termini** del metalinguaggio (ossia termini Prolog) che rappresentano espressioni linguistiche del linguaggio oggetto (il Prolog stesso).

Si e' scelta, per semplicita', come **relazione di naming** la funzione che "mappa" ogni

espressione linguistica in un termine di uguale struttura. In particolare, ogni costante e' rappresentata dalla costante stessa, ogni lettera funzionale e' rappresentata dalla lettera funzionale stessa ed ogni lettera predicativa e' rappresentata dalla lettera funzionale con il medesimo nome (in Prolog non e' richiesto che l'insieme delle lettere predicative e quello delle lettere funzionali siano disgiunti). I connettivi logici ":-" e "," sono rappresentati dalle lettere funzionali ":-" e ",". Ogni variabile che occorre in un termine, inoltre, non e' rappresentata da un termine costante ma rimpiazzata, all'interno della metarappresentazione di quel termine, da una metavariabile. In questo modo ogni espressione del linguaggio viene rappresentata da un termine uguale ad essa.

Per semplificare l'uso dei metapredicati, tuttavia, si e' scelto di rappresentare le teorie mediante nomi costanti, anziche' tramite termini strutturati. Questo, non solo comporta il vantaggio della sinteticita', ma offre anche la possibilita' di definire in modo semplice teorie che fanno riferimento a se stesse nelle proprie clausole, tramite il proprio nome (frasi autoreferenziali).

L'associazione tra il nome costante di una teoria e la sua descrizione e' mantenuta all'interno del metainterprete mediante il predicato

name_of(<nome_costante>,<descrizione_teoria>).

Ad ogni teoria (contenuta in un file) e' assegnato un nome costante dall'utente in fase di consultazione o dal sistema quando e' creata con una add_to o drop_from.

Il DB di programmi in MI_Prolog e' strutturato in teorie t1,..,tn.

Quando il goal e' un metapredicato, esso viene dimostrato nella metateoria Pr e le eventuali ridefinizioni dei metapredicati nelle altre teorie non vengono considerate.

Per gli altri predicati (non meta) non ci sono conflitti quando nomi uguali vengono usati in teorie differenti: a seconda del contesto in cui un goal e' dimostrato si usano le clausole della teoria opportuna.

I **built_in** di MI_Prolog sono i medesimi del C_Prolog; alcuni di essi, pero', sono stati ridefiniti nel metainterprete perche' caratterizzati in C_Prolog da una semantica extralogica o perche' inconsistenti con il resto del metainterprete.

I built_in assert e retract sono stati rimpiazzati da add_to e drop_from la cui semantica verra' descritta piu' precisamente nel seguito. I built_in listing(A), consult(File_List), Var is Expr, clause(H,B), ";", not(G) sono stati sostituiti, rispettivamente, dai predicati definiti nel metainterprete

m_listing(T,A), m_consult(L), m_is(X,Expr), m_clause(T,H,B), or(G1,G2), m_not(T,G),

con analoga semantica.

Alcuni dei nuovi predicati hanno, come argomento in piu', il nome della teoria nel cui contesto la chiamata del built_in deve essere effettuata. Il predicato m_consult(L) ha come argomento una lista di coppie (<nome_file>,<nome_teoria>) e una chiamata di tale predicato introduce nel DB, per ogni coppia, la teoria contenuta nel file <nome_file> ed associa alla sua descrizione, mediante il predicato name_of, il nome costante <nome_teoria>. Del predicato m_is si parlera' nel paragrafo III.5.

Infine il cut e' stato eliminato per preservare la semantica logica.

III.1.Semantica di MI_Prolog.

In questo capitolo viene presentata una semantica informale dei metapredicati introdotti in MI_Prolog. Essa risulta sostanzialmente simile a quella degli analoghi metapredicati proposti da Bowen e Kowalski, ma piu' dettagliata per quanto riguarda il trattamento delle variabili nella add_to e drop_from e l'interazione fra livello oggetto e meta.

La primitiva **demo(T,G,P)** ha successo se T e' il nome di una teoria o di una variabile istanziata al nome costante di una teoria; se il goal G e' dimostrabile in T e P e' la prova della dimostrazione.
G e' una lista (congiunzione) di predicati. P e' una lista di termini s(Subgoal,Rule), dove Subgoal e' il letterale selezionato nel passo di risoluzione, mentre Rule e' la clausola di input utilizzata.
Durante la dimostrazione di G, T diventa la teoria corrente. Dimostrato G, viene ripristinata, come teoria corrente, la teoria che era corrente quando il goal demo e' stato chiamato.
Ad esempio, dato il seguente DB:

```
t1:
   r(X):-demo(t2,[p(a),s(X)],[]),q(X).
   q(a).

t2:
   p(a).
   s(X):-p(X).
   q(b).
```

la query

```
   ?-demo(t1,[r(Z)],[]).
```

ha successo per Z=a. Il goal q(X) viene dimostrato nella teoria corrente t1.

L'argomento P e' usato anche per specificare se la dimostrazione deve essere condotta a livello oggetto o a livello meta. Quando P e' [], la prova della dimostrazione non e' richiesta; in tal caso, se il goal demo non e' chiamato nel contesto di una dimostrazione per la quale e' richiesta la prova e se G non contiene sottogoal che sono variabili, il goal G e' dimostrato direttamente a livello oggetto nella teoria T, con una maggiore efficienza rispetto alla dimostrazione tramite il metainterprete. Al livello oggetto, pero', non e' possibile l'uso delle variabili esistenziali e diventa importante l'ordinamento dei letterali nel body delle clausole (queste caratteristiche del linguaggio verranno illustrate nel seguito). E' l'utente che deve provvedere ad utilizzare il predicato demo(T,G,[]) nei suoi programmi solamente quando la dimostrazione del goal G nella teoria T non richiede le potenzialita' offerte dal metalivello (questo, ad esempio, e' sicuramente vero se T non contiene metapredicati). Solo in questo caso, infatti, il principio di riflessione giustifica il passaggio al livello oggetto per eseguire la dimostrazione.

Le linking rules e, in particolare, la regola che consente di dimostrare un goal demo direttamente al livello oggetto dovrebbero per correttezza essere implementate al livello

dell'interpretete del linguaggio oggetto, modificando l'interprete stesso. Non volendo, pero', modificare l'interprete, e' possibile utilizzare il metainterprete per realizzare una approssimazione delle linking rules. Per passare a dimostrare un goal al livello oggetto, infatti, utilizzando le primitive extralogiche del Prolog, si e' introdotta nel metainterprete una clausola del tipo:

```
demo(T,G,[]):-in_DB(T),name(G,G1),G1.
```

dove in_DB e' un predicato che inserisce nel DB la teoria rappresentata dal termine T. Come si e' evidenziato nel paragrafo II, non e' possibile definire nella logica del I ordine un predicato name che permetta di associare ogni formula alla sua rappresentazione, in quanto una formula non puo' essere argomento di un predicato. Per questo, non e' possibile simulare le linking rules nel metainterprete rimanendo nella logica del I ordine.

La primitiva **add_to(T1,List,Arg,T2)** ha successo se T1 e' il nome costante di una teoria oppure una variabile istanziata al nome di una teoria e se T2 e' una variabile non istanziata. Se la primitiva ha successo, viene creata una nuova teoria, che contiene tutte le clausole di T1 e la clausola Arg e ad essa e' assegnato un nome costante unico, con cui viene istanziata la variabile T2.
Formalmente (tralasciando List):

$$Mt \vdash_L \text{add_to("T1","Clause","T2")} \quad \text{sse} \quad T2 = T1 \cup \{Clause\}.$$

Tutte le variabili della clausola Arg aggiunta che occorrono nella lista di variabili List sono universalmente quantificate e hanno "scope" interno alla add_to; le altre sono dette esistenzialmente quantificate, ed hanno come scope l'intera clausola in cui il predicato add_to occorre. Nel seguito sara' descritto come le variabili esistenziali sono state introdotte.
In MI_Prolog le variabili si intendono quantificate universalmente nella clausola in cui occorrono, come in Prolog, con la sola eccezione per le variabili che occorrono nella lista List delle primitive add_to e drop_from.

La primitiva **drop_from(T1,List,Arg,T2)** ha successo se T1 e' il nome costante di una teoria oppure una variabile istanziata al nome di una teoria, se tale teoria contiene una clausola che unifichi con Arg e se T2 e' una variabile non istanziata.
Se la primitiva ha successo, viene creata una nuova teoria che contiene tutte le clausole di T1 tranne la clausola che unifica con Arg e ad essa viene assegnato un nome costante unico con cui viene istanziata la variabile T2.
La clausola che viene eliminata e' la prima che unifica con Arg; le variabili di tale clausola che corrispondono alle variabili universali di Arg, devono essere anch'esse universali.

La primitiva **current(T)** ha successo se T unifica con il nome della teoria corrente. L'informazione su quale sia la teoria corrente, viene mantenuta nella metateoria Pr.

Il significato della chiamata di una add_to o di una drop_from, quando la clausola da aggiungere o togliere e' un fatto (una formula atomica), non e' quello di rendere rispettivamente vero o falso il fatto nella teoria creata, ma quello di manipolare

sintatticamente una teoria per crearne un'altra con una clausola in piu' o in meno.
Le teorie sono infatti viste come oggetti sintattici, insiemi di assiomi, che il metalivello tratta come oggetti del I ordine mediante i metapredicati add_to e drop_from.

Il motivo per cui e' necessario che il quarto argomento delle primitive add_to e drop_from sia una variabile non istanziata e' quello di evitare i conflitti fra i nomi delle teorie create: l'utente non ha la possibilita' di specificare il nome costante della nuova teoria creata dalla add_to o dalla drop_from; sono le due primitive stesse che generano ad ogni chiamata un nome nuovo per la teoria creata e con esso istanziano la variabile T2.

III.2.Definizione del metainterprete

Il metainterprete e' stato introdotto definendo il metapredicato demo, che secondo l'interpretazione procedurale dei programmi logici, puo' essere considerato un interprete del Prolog scritto in Prolog.
Il metapredicato demo e' definito nel seguente modo:

```
demo(Theory,Goal,[Reason|Rest_Proof])
    :-
    select(Goal,SubGoal,Rest_Goals),
    react(Theory,SubGoal,Reason,Continuation_Goals),
    merge(Continuation_Goals,Rest_Goals,New_Goal),
    demo(Theory,New_Goal,Rest_Proof).
```

dove:

demo: dimostra il goal in Theory e ne fornisce la prova.

select: seleziona il primo sottogoal della lista Goals; Rest_Goals e' la lista dei Goals rimanenti.

react: cerca fra le clausole della teoria specificata una regola il cui head sia unificabile con SubGoal. Se tale regola viene trovata, la variabile Continuation_Goals viene istanziata alla lista dei predicati che costituiscono il body della regola. La variabile Reason assume il valore di una struttura del tipo s(Subgoal, Rule), che associa al sottogoal la regola usata per dimostrarlo.

merge: concatena la lista Continuation_Goals e la lista Rest_Goals. La lista risultante, Newlist, contiene tutti i goal ancora da dimostrare. Su tale lista di goal viene chiamato il predicato demo.

Per la definizione degli altri metapredicati si veda la tesi di laurea [GiMu87]).

III.3.MI_Prolog preserva la semantica logica e dichiarativa.

La primitiva assert del Prolog viola la semantica logica, in quanto:

(a) il significato di una clausola in cui occorre una assert dipende dall'ordine in cui i letterali compaiono nel body. Esempio:

(1) ?-assert(p(a)),p(a).

(2) ?-p(a),assert(p(a)).

la (1) ha successo, mentre la (2) non ha successo se il DB non contiene p(a);

(b) non si fa backtracking sulla assert: cio' che e' stato aggiunto al DB mediante una assert non viene eliminato su backtracking;

(c) il significato delle variabili che occorrono nello scope di una assert e' differente da quello di tutte le altre variabili. Infatti una variabile nella assert non istanziata al momento della chiamata e' considerata universalmente quantificata, mentre una variabile istanziata e' considerata esistenzialmente quantificata.

Esempio 1. Dato il programma Prolog

```
r(X):-assert(p(X)),p(a),q(X).
q(b).
```

la query ?-r(X) ha successo con X=b, mentre la query ?-r(b) fallisce.

In MI_Prolog questi problemi sono stati risolti, poiche' si possono trattare le variabili esistenzialmente quantificate (cf.par.III.4); la semantica di una clausola e' indipendente dall'ordinamento dei letterali nel suo body (cf.par.III.5); si puo' fare backtracking sul predicato add_to e sugli altri metapredicati.

III.4.Variabili esistenzialmente quantificate.

MI_Prolog consente di introdurre in una teoria, mediante la primitiva add_to, delle clausole le cui variabili sono quantificate esistenzialmente; l'utilita' del disporre di variabili esistenziali viene evidenziata in [BoWe85, BWei85], in cui non e' pero' affrontato il problema di come esse possano essere trattate.

MI_Prolog consente di specificare a livello sintattico quali sono le variabili universali e quali le esistenziali che occorrono in una add_to e in una drop_from e permette un loro adeguato trattamento eliminando ogni ambiguita'.

Quando il goal selezionato per essere dimostrato e'

add_to(T1,List,Arg,T2)

si possono presentare tre casi, a seconda che la clausola Arg da aggiungere contenga (a) solo variabili universali, (b) solo variabili esistenziali oppure (c) entrambe. Il parametro List specifica quale caso si sta trattando; esso contiene tutte e sole le variabili universali presenti nella clausola Arg.

(a) Se la clausola da aggiungere contiene solamente variabili universali o/e esistenziali istanziate, come nel caso add_to(t1,[X],(p(X):-q(X)),T2), essa viene aggiunta nel DB.

(b) Se la clausola contiene solamente variabili esistenziali alcune delle quali non istanziate, come nel caso add_to(t1,[],q(X,Y),T2), la sua metadescrizione viene inserita in una lista di supporto, interna al metainterprete, in modo da bloccare le istanziazioni delle variabili.

(c) Se variabili esistenziali non istanziate ed universali sono compresenti nella clausola da aggiungere, come nel caso add_to(t1,[Y],q(X,Y),T2), la chiamata di tale primitiva non puo' essere eseguita fintanto che tutte le variabili esistenziali non sono state istanziate: il goal viene ritardato e un altro goal e' selezionato.

Questa soluzione, comunque, non garantisce che il trattamento delle variabili esistenziali sia completo, perche' puo' succedere che, in questo ultimo caso, le variabili esistenziali non vengano mai istanziate e quindi il goal non possa essere eseguito.

Una teoria MI_Prolog e' quindi costituita, in generale, da un insieme di clausole con variabili universalmente quantificate (contenute nel DB) e da un insieme di clausole contenenti variabili quantificate esistenzialmente (contenute nella lista di supporto). Quando una clausola di queste ultime viene utilizzata in una dimostrazione e le sue variabili vengono istanziate, la istanziazione e' permanente a meno di un successivo backtracking.

Il programma nell'esempio 1 puo' essere riscritto in MI_Prolog, eliminando le ambiguita' sul significato delle variabili nel seguente modo.

Sia t0 la teoria:

```
r(X):-current(T1),
      add_to(T1,[],p(X),T2),
      demo(T2,[p(a),q(X)],[]).
q(b).
```

La variabile X e' stata considerata quantificata esistenzialmente.
Entrambe le query

```
?-demo(t0,[r(X)],P).
?-demo(t0,[r(b)],P).
```

falliscono.

Se si considera la variabile X (che occorre come argomento della assert nell'esempio 1) quantificata universalmente, allora la teoria t0 deve essere riscritta nel seguente modo:

```
r(X):-current(T1),
      add_to(T1,[Y],p(Y),T2),
      demo(T2,[p(a),q(X)],[]).
q(b).
```

In tal caso le due query:

```
?-demo(t0,[r(X)],P).
?-demo(t0,[r(b)],P).
```

hanno entrambe successo; la prima con la sostituzione X=b.

La primitiva drop_from cerca la clausola da eliminare prima nel DB e poi nella lista di clausole esistenziali.

III.5.Commutativita' della virgola.

Si vuole che il significato delle clausole sia in generale indipendente dall'ordine dei letterali nel loro body, requisito, questo, necessario affinche' il linguaggio possa avere una semantica

dichiarativa. Per ottenere questo e' stata utilizzata la tecnica di ritardare i goal.

Come si e' gia' visto, vengono ritardate le chiamate della primitiva add_to(T1,List,Arg,T2), quando la clausola Arg contiene occorrenze di variabili universali e di variabili esistenziali non ancora istanziate. E' per questo indifferente che un predicato add_to segua o preceda nel body di una clausola il predicato che istanzia una sua variabile esistenziale.

```
?- X=a,add_to(t1,[Y],p(X,Y),T2).
?- add_to(t1,[Y],p(X,Y),T2),X=a.
```

sono queries che hanno la stessa semantica.
Per ritardare un goal ci si avvale della presenza esplicita nel metainterprete della lista dei goal da dimostrare; da questa il predicato select, seleziona, nell'ordine, il primo goal che puo' essere risolto, lasciando nella lista i goal che devono essere ritardati.
Perche' il connettivo logico "and" (rappresentato dalla virgola nel body delle clausole) sia commutativo, oltre ai goal add_to(), devono poter essere ritardati i goal demo() e drop_from().
Ritardare l'esecuzione di un goal add_to(Ti,L,Clause,Tj), ritarda infatti la creazione di una nuova teoria e l'istanziazione della variabile Tj al nome di tale teoria; risulta pertanto impossibile, fintanto che il goal add_to() non viene dimostrato, dimostrare un qualsiasi goal che faccia riferimento a Tj. E' quindi necessario potere ritardare i goal demo(T,G,P), add_to(T,List,Arg,T2), drop_from(T,List,Arg,T2) e tutti i built_in che hanno come argomento una teoria T fino a quando T non e' stata creata. La possibilita' di ritardare un goal demo rende indifferente l'ordine dei predicati demo() e add_to() nel body delle clausole.
Infine, devono essere ritardati il goal m_not(T,G) (il predicato m_not e' stato introdotto in sostituzione del not del Prolog; si veda tesi di laurea [GiMu87] cap.V) e il goal m_is(Var,Expr) quando G o, rispettivamente, Expr contengono variabili non istanziate.
Se nella lista dei goal da dimostrare non ne esiste alcuno che possa essere selezionato, il metainterprete non segnala il fallimento, ma verifica se nella lista c'e' un goal m_not (m_is) in cui G (Expr) contiene variabili non istanziate, oppure un goal add_to in cui Arg contiene variabili universali ed esistenziali non istanziate; se trova un tale goal, allora risponde "non so", viceversa "no". In questo modo si ottiene la correttezza delle dimostrazioni condotte al metalivello.

III.6.Backtracking.

I metapredicati demo(), add_to() e drop_from() sono stati definiti in modo che su di essi si possa fare backtracking. Per quanto concerne le primitive add_to e drop_from, su backtracking viene eliminata la descrizione della teoria creata (e della clausola aggiunta nel caso della primitiva add_to).
Fare backtracking su una chiamata della primitiva demo significa ripristinare, come teoria corrente, quella che era tale prima della chiamata, rendendola disponibile nel DB. Questa operazione di ripristino della teoria corrente e' compiuta anche in caso di successo del predicato demo.

IV. RAGIONAMENTO IPOTETICO.

Per Ragionamento Ipotetico si intende il ragionare sotto certe ipotesi, supponendo queste vere.
Spesso quando si ha a che fare con problemi su domini reali e' necessario **generare assunzioni**, sulla base delle quali si fanno deduzioni di nuovi enunciati che possono confermare o contraddire l'assunzione fatta. Generare la soluzione completa di un problema puo' richiedere:

-assunzioni temporanee su soluzioni parziali;
-assunzioni sui vari livelli di descrizione.

Per questo e' utile disporre di un linguaggio logico che permetta di fare Ragionamento Ipotetico.

In un linguaggio logico, fare Ragionamento Ipotetico significa dare una lettura ipotetica della implicazione, lettura che differisce da quella della logica classica (come e' stato dimostrato in [GaRe84, Gabb85], e' quella della logica intuizionista).

Un **goal ipotetico** e' un goal del tipo:

?-A→B

che significa:

"Aggiungi A al DB e dimostra B nel nuovo DB".

I goal che seguono un goal ipotetico vengono dimostrati nel DB di partenza e non in quello modificato in seguito all'esecuzione di tale goal.
Esempio

?-(A→B),C

il goal C viene dimostrato nel DB precedente all'aggiunta di A.

Per fare ragionamento ipotetico non bastano le clausole di Horn; sono necessarie clausole piu' generali che possano, ad esempio, avere nel body delle **implicazioni ipotetiche**, anziche' solamente letterali (ad esempio la clausola (D∧(A→B))→C).

L'implicazione ipotetica potrebbe essere realizzata in Prolog usando le primitive assert e retract nel modo seguente:

A→B:-assert(A),B,retract(A),!.
A→B:-retract(A),fail.

Si presentano, pero', dei problemi quando in un goal ipotetico occorrono delle variabili, come ad esempio in

(1) ?-A(X)→B(X)

Esse devono essere considerate quantificate esistenzialmente. La lettura ipotetica del goal (1) e' percio':

"Esiste un X tale che, se A(X) e' aggiunto al DB, il goal B(X) ha successo".

In Prolog, con l'implementazione dell'implicazione mostrata sopra (non disponendo delle variabili esistenziali), nel dimostrare il goal (1), viene asserito ∀XA(X) nel DB e quindi

regola di derivazione per le implicazioni. Data una clausola di N_Prolog, usando la primitiva imply si puo' scrivere in MI_Prolog una clausola corrispondente (con la medesima semantica).

La primitiva imply e' stata definita nel metainterprete mediante le primitive **add_to** e **drop_from**.
La possibilita' di trattare le variabili esistenziali nel metainterprete permette di fare ragionamento ipotetico in MI_Prolog senza dovere implementare un linguaggio ex_novo, come invece e' stato fatto per N_Prolog.
La suddivisione del DB in teorie permette, tra l'altro, di evitare il problema dei side_effects nella dimostrazione dei goal ipotetici: si crea una nuova teoria uguale alla corrente a cui si aggiunge la lista di clausole A e in cui si cerca di dimostrare B, quindi si ritorna con la computazione nella teoria di partenza.
Il linguaggio Multilog [KaGr86] permette di fare ragionamento ipotetico, ma lascia irrisolto il problema delle variabili esistenziali nei goal di tipo ipotetico.

V. CENNI SULL'IMPLEMENTAZIONE DI MI_PROLOG.

Il linguaggio MI_Prplog e' stato implementato sul C_Proplog. Non si intende in questo paragrafo entrare nei dettagli dell'implementazione di MI_Prolog, ma semplicemente si vogliono fare alcune considerazioni generali su di essa.

In primo luogo si deve osservare che la scelta della relazione di naming e, in particolare, la scelta di non rappresentare le variabili con dei termini ground, ma di sostituirle con metavariabili, ha permesso di definire il predicato demo utilizzando l'algoritmo di unificazione del Prolog stesso, consentendo quindi una maggiore efficienza nelle dimostrazioni al metalivello. Viceversa, sarebbe stato necessario mantenere esplicitamente, come argomento del predicato demo, la lista delle sostituzioni delle variabili e definire al metalivello un algoritmo di unificazione.

Al livello meta al nome di ogni teoria viene associata, come si e' detto, la descrizione della teoria stessa mediante il predicato name_of.
Per poter dimostrare un goal in una teoria al livello oggetto, e' pero' necessario che la teoria sia presente nel DB. Se si dispone di unico DB, volendo fare una dimostrazione al livello oggetto in una teoria T, si deve sostituire T alla teoria correntemente nel DB. E' utile, per ridurre le operazioni di sostituzione delle teorie nel DB, disporre di un DB strutturato in cui possano essere memorizzate piu' teorie contemporaneamente.
MI_Prolog e' stato implementato utilizzando i DB con chiave del C_Prolog, che permettono di associare nomi costanti ad insiemi di clausole.
Ad ogni teoria iniziale vengono associati un DB che ha come chiave il suo nome ed un albero che rappresenta i legami fra le teorie create con add_to e drop_from a partire dalla teoria iniziale stessa (i nodi dell'albero sono teorie e la radice e' la teoria iniziale).
Le teorie di ogni albero hanno a disposizione un unico DB con chiave, quello associato alla radice; in tale DB e' sempre contenuta una sola delle teorie dell'albero, inizialmente proprio

viene dimostrato B(X); fare cio', pero', non corrisponde alla lettura ipotetica del goal.
Infatti, dato il DB:

```
p(b).

q(X):-
    p(X),
    s(X),
    s(a).
```

e il goal

```
?-s(X)→q(X).
```

se si dimostra la query ipotetica aggiungendo al DB la clausola $\forall X s(X)$ e poi dimostrando ?-q(X), la clausola aggiunta viene utilizzata sia per dimostrare s(b) che s(a); cio' differisce dalla lettura ipotetica del goal.

Nella logica classica le due formule:

(2) $P \rightarrow \exists X(A(X) \rightarrow B(X))$
(3) $P \wedge \forall X A(X) \rightarrow \exists X B(X)$

sono equivalenti, mentre non lo sono nella logica intuizionista. La lettura ipotetica della query (1) su un database P coincide con l'interpretazione della formula (2) nella logica intuizionista. Viceversa, il trattamento in Prolog di tale query corrisponde alla (3) ed e' quindi scorretto.
E' percio' necessario potere disporre delle variabili esistenziali per trattare correttamente i goal di tipo ipotetico.

Il linguaggio N_Prolog [GaRe84, Gabb85] realizza il Ragionamento Ipotetico estendendo il Prolog sintatticamente (le clausole possono avere nel body implicazioni ipotetiche) e introducendo una regola di deduzione per le implicazioni in aggiunta a quelle per le congiunzioni e per gli atomi. In [GaRe84] vengono dimostrate la correttezza e la completezza di N_Prolog rispetto alla logica intuizionista.

Al contrario in MI_Prolog ci si e' basati sulle possibilita' offerte dalla presenza di un metalivello per poter fare Ragionamento Ipotetico. Piu' precisamente si e' introdotto un metapredicato **imply(A,B)**, la cui semantica e' la seguente:

> una chiamata della primitiva imply(A,B) ha successo se il goal B puo' essere dimostrato nella teoria corrente a cui sono state aggiunte le clausole di A, dove A=[A1,..,An].

Le variabili contenute nella lista di clausole A sono trattate come variabili quantificate esistenzialmente.
Introdurre la imply nell'MI_Prolog, equivale ad introdurre la implicazione ipotetica e la

la radice.
Quando si vuole fare una dimostrazione a livello oggetto in una teoria T, se questa non e' presente nel DB con chiave associato al suo albero, e' necessario sostituire in tale DB la teoria T alla teoria T1 che lo occupa (che e' certamente una teoria del medesimo albero), modificando T1 sino ad ottenere T. Le operazioni di modifica vengono eseguite percorrendo l'albero a cui appartengono T e T1, che fornisce le informazioni sui legami che intercorrono fra di esse.
Questa scelta e' motivata dal fatto che due teorie T e T1 appartenenti allo stesso albero hanno presumibilmente la maggior parte delle clausole in comune e quindi le operazioni di modifica di T1 in T non sono numerose.
In questo modo si e' cercato di realizzare un compromesso fra il consumo di spazio di memoria ed il tempo necessario ad introdurre una teoria nel DB, quando si vuole fare una dimostrazione in essa al livello oggetto.

CONCLUSIONI.

MI_Prolog e' un linguaggio che estende il Prolog mediante l'introduzione di un metalivello. Tale linguaggio e' inerentemente meno efficiente del Prolog, in quanto la presenza di un metainterprete introduce un livello di simulazione nella dimostrazione dei goal. Nella implementazione attuale in C_Prolog, l'inefficienza e' inoltre dovuta al fatto che non si dispone di un DB strutturato per memorizzarvi le teorie in cui fare le dimostrazioni al livello oggetto (ci sono i DB con chiave, ma non le primitive per operare in modo semplice su di essi). Per migliorare l'efficienza, MI_Prolog potrebbe essere implementato convenientemente sopra Env_Prolog [MaRo87], un Prolog esteso che fornisce meccanismi di strutturazione del DB (si possono definire moduli) e permette di rappresentare clausole e programmi come dati (utilizzando una rappresentazione interna uniforme per i termini, le clausole ed i programmi).

Seguendo l'approcio di Bowen e Kowalski, come si e' visto, si puo' estendere un linguaggio del I ordine con dei metapredicati, pur rimanendo nella logica del I ordine, grazie al fatto che ogni espressione linguistica puo' essere rappresentata mediante un termine. E' tuttavia possibile, da un altro punto di vista, considerare i simboli demo, add_to e drop_from come operatori modali, anziche' lettere predicative del linguaggio. In tal caso non e' piu' necessario disporre della rappresentazione di ogni espressione del linguaggio mediante un termine: demo, add_to e drop_from sono operatori che vengono applicati direttamente alle formule del linguaggio. Inoltre, e' possibile dare al linguaggio ottenuto introducendo questi operatori modali una semantica dei mondi possibili, alla Kripke.

RIFERIMENTI BIBLIOGRAFICI

[BoKo82] Bowen K.A.,Kowalski R.A.: "Amalgamating Language and Metalanguage in

Logic Programming". In **Logic Programming** (K.L.Clark and S. Tarlund, Eds.) 1982, 153-172.

[Bowe85] Bowen K.A.: "Meta_Level Programming and Knowledge Representation". In **New Generation Computing** 1985:3:359-383.

[BWei85] Bowen K.A., Weinberg T.: "A Meta_Level Extension of Prolog". In **Proceedings of the 2nd IEEE International Symposium on Logic Programming**, Boston, USA, July, 1985.

[GaRe84] Gabbay D.M., Reyle N.: "N_Prolog: An Extension of Prolog with Hypotetical Implications-I". In **Journal of Logic Programming**, 1984:4:319-355.

[Gabb85] Gabbay D.M.: "N_Prolog: An Extension of Prolog with Hypotetical Implications-II. Logical foundations, and negation as failure". In **Journal of Logic Programming**, 1985:4:251-283.

[GiMu87] Giordano L., Murgia I.:"L'uso della metaconoscenza nella programmazione logica", tesi di laurea, Dipartimento di Informatica di Torino, 1987.

[KaGr86] Kauffman H., Grumbach A.: "MULTILOG: MULTiple worlds in LOGic programming". In **Proceedings of the 7th European Conference on Artificial Intelligence**, July 1986.

[Kowa83] Kowalski R.: "Logic Programming". In IFIP 1983, R.E.A. Mason ed., Elsevier Science Pub. (1983), pp. 133-145.

[MaRo87] Martelli A., Rossi G.F.:"PROSE: un ambiente di programmazione per il Prolog". In **Atti del convegno nazionale sulla programmazione logica**, Torino, Maggio 1987.

[NaAb87] Nait Abdallah M.A.:"Logic Programming with ions". In Thomas Ottman (Ed.), Automata Languages and Programming, 14th International Colloquium, Karlsruhe, 1987. Proceedings in **Lectures Notes in Computer Science** vol. 267, Springer-Verlag.

Correttezza e Completezza della *Valutazione Parziale* di Programmi Logici

Laura Fanti*, Giorgio Levi+, e Sandra Zanobetti*

*Tecsiel S.p.A, via S.Maria 19, 56100 Pisa
+Dipartimento di Informatica, Università di Pisa

Sommario: In questo articolo viene presentato un algoritmo di *valutazione parziale* per programmi HCL (*Horn Clauses Language*) basato sulla *resolution SLD*, e ne viene dimostrata in maniera formale la correttezza.

In particolare in relazione al trattamento di programmi ricorsivi con computazioni infinite viene messa in rilievo la necessità di assicurare la terminazione dell'algoritmo in un numero finito di passi e di garantire contemporaneamente che esso preservi l'equivalenza semantica tra il programma originale e quello generato: questi problemi, non risolubili in generale, vengono risolti rispetto a classi di programmi HCL.

1. Introduzione alla Valutazione Parziale

La valutazione parziale è una tecnica di trasformazione di programmi [Ershov-77], che si basa sul teorema S-m-n di Kleene:

"Data una funzione $f=\lambda x_1.....x_n.\varepsilon$ e k valori costanti ($k \leq n$) $a_1,.....,a_k$ è possibile calcolare una funzione $f'=\lambda x_{k+1}.....x_n.\varepsilon'$ tale che:
$f(a_1,.....,a_k,x_{k+1}.....x_n)=f'(x_{k+1}.....x_n)$".

Il risultato della "valutazione parziale" di una funzione f è quindi una funzione f' che specializza f ed é più efficiente per i valori specifici $a_1,.....,a_k$.

La valutazione parziale di una funzione f in definitiva non é che un caso di computazione in cui non tutti gli argomenti (al limite nessuno) sono noti. Il risultato di tale computazione in generale sarà ancora una funzione f' che ha come argomenti quelli non noti al momento della valutazione di f.

La valutazione parziale richiede un'analisi dinamica del programma sorgente, essendo caratterizzata dai seguenti aspetti: