METODI DI RISCRITTURA DI CLAUSOLE DI HORN PER LA VALUTAZIONE DI PREDICATI RICORSIVI

Maria Luisa Sapino

Dipartimento di Informatica, Universita' di Torino
c. Svizzera 185 - 10149 TORINO (ITALY) - tel. + (39)(11)755677

Sommario.

I metodi di riscrittura Counting ed Alexander sono algoritmi che ricevono in input un programma logico (costituito da un insieme di clausole di Horn e da un goal, e pensato per essere eseguito secondo una strategia backward, Prolog-like) e producono in output un altro programma logico, equivalente a quello dato, per il quale risulti efficiente una strategia di esecuzione forward (basata sull'operatore di "saturazione di un insieme di clausole").

In questo lavoro si esaminano i due metodi e, dopo aver descritto una corrispondenza tra i linguaggi di programmazione logici e quelli imperativi convenzionali (Pascal-like), si introduce la semantica di una macchina a stack per l'esecuzione delle procedure in linguaggio convenzionale. Di questa macchina sono noti sia il comportamento sia le proprieta', e la si puo' quindi utilizzare come ambiente di riferimento per lo studio del comportamento e delle proprieta' dei programmi logici prodotti dalle riscritture Counting ed Alexander. In particolare ci si concentra sulle proprieta' di terminazione e sullo studio degli eventuali motivi di inefficienza dovuti alla configurazione dei dati.

1. Introduzione.

Le relazioni presenti in una base di dati logica possono essere distinte in *relazioni di base* (esplicitamente memorizzate in una base di dati esterna) e *relazioni derivate o virtuali*, definite in termini di relazioni di base e/o, ricorsivamente, di relazioni derivate.

In presenza di relazioni virtuali ricorsive, si pone il problema di trovare delle strategie efficienti per calcolare le intere relazioni e per calcolare eventuali sottinsiemi delle stesse.

Gran parte della letteratura sulle basi di dati logiche affronta il problema di una comunicazione efficiente con la base di dati esterna, soprattutto per i casi in cui si debbano valutare relazioni ricorsive [Banc86, Hens81, Rom85]. Tra le varie proposte hanno riscosso particolare interesse i cosiddetti *metodi di riscrittura per programmi logici ricorsivi*, algoritmi che "riscrivono" un insieme di clausole di Horn concepite per una strategia di valutazione backward, Prolog-like, in un altro insieme di clausole che possano essere efficientemente valutate con una strategia forward.

In questo lavoro si esaminano i metodi di riscrittura Counting [Banc85], [Sacc86], ed Alexander [Rohm85], [Rohm86]. Un metodo di riscrittura e' costituito fondamentalmente da due componenti: un algoritmo di conversione, ed una strategia di computazione per

valutare la risposta richiesta. In [Rohm86] la strategia di valutazione forward relativa al metodo Alexander e' definita a partire dall'operatore di *saturazione di un insieme di clausole*; in questo lavoro si adotta la stessa strategia anche per il metodo Counting. Sotto queste ipotesi, si confrontano i due metodi di riscrittura con l'ambiente logico originale, utilizzando una macchina a stack come ambiente comune di riferimento. Questo confronto, suggerito in [Demo87], e' qui descritto in dettaglio fino a dimostrare che le riscritture Counting ed Alexander di un programma logico sono equivalenti, rispettivamente in modo forte ed in modo semplice, all'esecuzione sulla macchina a stack delle procedure in linguaggio convenzionale ad esso corrispondenti.

Nel paragrafo seguente si introduce innanzi tutto un'interpretazione procedurale di un programma logico (per definirne la semantica operazionale), quindi, nel terzo paragrafo, si descrive la semantica di una macchina a stack per l'esecuzione delle procedure associate ai predicati di una base di dati logica. Tale esecuzione puo' essere completamente rappresentata mediante un albero ordinato (*albero dell'evoluzione della stack*), le cui caratteristiche e proprieta' sono presentate nel quarto paragrafo. Nel paragrafo 5 si utilizza quest'albero come strumento per la formulazione di una rilettura dei metodi Counting ed Alexander, allo scopo di evidenziarne, nel successivo, le proprieta' di correttezza, completezza e terminazione, oltre agli eventuali motivi di inefficienza.

2. Semantica operazionale di un programma logico: interpretazione procedurale.

In questo paragrafo si presenta la somiglianza tra linguaggi logici e linguaggi di programmazione convenzionali, considerando le clausole come definizioni di procedure ed i goals come chiamate di procedure, secondo [vanE76]. In particolare si segue il metodo descritto in [Demo86], dove questo paradigma e' integrato con una trascrizione secondo l'algebra relazionale dei corpi delle clausole.

Secondo tale metodo, dati un programma logico ed un goal, si deve innanzi tutto esaminare la successione dei richiami ricorsivi da effettuarsi per la valutazione del goal stesso. Ogni richiamo e' caratterizzato dal nome di un predicato e dall'insieme delle variabili in esso istanziate, e ad ogni coppia <nome-predicato, variabili istanziate> si deve associare una procedura.
Ad esempio, dati la relazione same-gen definita dalle clausole

(1) same-gen(X,Y):- Y is X.
same-gen(X,Y):- parent(X,V), same-gen(V,W), parent(Y,W).

ed il goal same-gen(a,Y), il metodo deriva la seguente procedura same-gen-bf, in cui il suffisso bf sta per <bound,free>, dal momento che, nel goal considerato, le variabili <X,Y> sono rispettivamente legata ad 'a' e libera.
La procedura derivata e' unica poiche' le variabili istanziate nel predicato ricorsivo nel corpo della clausola occupano le stesse posizioni di quelle che sono istanziate nella testa della stessa.


```
    procedure same-gen-bf (In(X):In-relation, Out(Y):Out-relation);
      begin
1       subanswer1(Y):= In(X);
2       subanswer21(V):= π_Par-name(parent(Son-name,Par-name)|/|_1.1=2.1 In(X));
3       if subanswer21(V) ≠ {} then
                same-gen-bf(subanswer21(V),subanswer22(W))
4             else  subanswer22(W):= {};
5       if subanswer22(W) ≠ {} then
                subanswer23(Y):= π_Son-name(parent(Son-name,Par-name)|/|_1.2=2.1 subanswer22(W))
6             else  subanswer23(Y):= {};
7       Out(Y):= subanswer1(Y) ∪ subanswer23(Y);
      end;
```

I dati di input alla procedura same-gen-bf sono costituiti dalla relazione In(X) il cui schema contiene gli attributi corrispondenti alle variabili istanziate nella query. In questo caso In contiene soltanto un attributo del tipo X, essendo X l'attributo istanziato nel goal same-gen(a,Y). Il risultato della procedura e' una relazione Out(Y), il cui schema e' complementare a quello della relazione di input In (in generale e' l'intero schema della relazione ricorsiva coinvolta nella query perche', per il seguito della computazione, puo' servire conservare le tuple dell'intera relazione calcolata). Le variabili interne della procedura sono le relazioni subanswer1, subanswer21, subanswer22 e subanswer23 che corrispondono rispettivamente all' unico predicato nel corpo della prima clausola in (1), e al primo, al secondo e al terzo predicato nel corpo della seconda clausola in (1).
La procedura same-gen-bf chiama se stessa ricorsivamente finche' non esista piu' nessun valore di input X per il quale l'informazione sul genitore sia presente nella base di dati. La condizione di terminazione certamente si verifichera', nell'ipotesi che la relazione parent sia aciclica, essendo la base di dati finita.
La computazione della procedura same-gen-bf avviene secondo il modello di un "interprete Prolog orientato agli insiemi", cioe' si comporta come un interprete Prolog che colleziona simultaneamente tutte le risposte ad un goal piuttosto che raccoglierle una per volta. L'istanziazione stessa del goal e' costituita da una relazione, invece che da una singola tupla. Quando viene formulato il goal same-gen(a,Y), si invoca la procedura same-gen-bf per valutare la risposta con In(X)={<a>}.
Il simbolo $|/|_{1.1=2.1}$ che compare nell'istruzione 2 della procedura indica un equijoin sul primo attributo della prima (a sinistra di |/|) relazione e il primo attributo della seconda (a destra) relazione; π e' l'operatore di proiezione dell'algebra relazionale.

Un altro esempio, piu' generale, e' l'insieme S di clausole mostrate in (2). Tale esempio e' tratto da [Sacc86].

(2) P(X,Y):-B1(X,X_1),Q(X_1,Y),B2(X,X_2),Q(X_2,Y),B3(Y,Z).
Q(X,Y):-B4(X,Z),P(Z,Y).
P(X,Y):-B5(X,Y).

Dato il goal P(a,Y), il metodo suddetto deriva quattro procedure Pascal-like: p-bf, q-bf, q-bb e p-bb Infatti in una strategia di valutazione da sinistra a destra, il secondo predicato Q nella prima clausola di (2) ha entrambe gli argomenti X_2 e Y legati, e questi rendono legati

entrambe gli argomenti di P nella seconda clausola.

Se invece di attenersi ad una valutazione sequenziale si pensa ad una valutazione parallela dei diversi predicati ricorsivi presenti nel corpo di una stessa clausola del programma logico, si ottiene un'interpretazione procedurale diversa dalla precedente.
Una scelta di questo tipo consente di ridurre i tempi di risposta, ma puo' richiedere il calcolo di relazioni piu' grandi di quelle strettamente necessarie per la valutazione di un goal. Sotto queste ipotesi le procedure associate al programma logico (2) si riducono alle seguenti p-bf e q-bf:

```
procedure p-bf(In(X):In-rel ; Out(Y):Out-rel);
begin
   1  sub2(Y):= π_Y(In(X) |/| B5(X,Y));
   2  sub01(X1,X2):=π_{X1,X2}(In(X) |/| B1(X,X1) |/| B2(X,X2));
      if sub01(X1,X2) ≠ {} then
      begin
   3      q-bf(π_{X1}sub01(X1,X2), sub02(Y))
   4      q-bf(π_{X2}sub01(X1,X2), sub03(Y))
   5      sub04(Y):= π_Y( sub02(Y) |/| sub03(Y) |/| B3(Y,Z))
      end
   6    else sub04:={};
   7  Out(Y):= sub2(Y) ∪ sub04(Y);
end;


procedure q-bf(In(X):In-rel ; Out(Y):Out-rel);
begin
   1  sub11(Z):= π_Z(In(X) |/| B4(X,Z));
   2  if sub11(Z) ≠ {} then p-bf(sub11(Z), Y)
   3     else Out(Y):= {};
end;
```

Figura 1

3. Predicati di una base di dati logica e semantica di una macchina a stack.

Per l'esecuzione delle procedure scritte in linguaggio convenzionale, si puo' pensare a una macchina a stack. In questo contesto ogni chiamata di procedura ha il proprio "record di attivazione" e tutte le aree delle variabili sono impilate sulla stack secondo una politica first-in-last-out.
Si consideri nuovamente il goal P(a,Y) per la relazione P(X,Y) definita in (2). L'esecuzione della corrispondente procedura ricorsiva p-bf sulla semplice base di dati in Figura 2 e' rappresentata dalla evoluzione della stack mostrata nella medesima figura.

Base di dati:

B1	B2	B3	B4	B5
a---->s	a---->t	l---->w	a---->b	b---->f
k---->t	e---->a	f---->y	s---->g	g---->k
z---->a		k---->r	t---->d	d---->k
		m---->n		a---->m

Storia dell'evoluzione della stack:

```
        ------------------------
        | In(X) = {a}          |
p-bf    | sub2(Y) = {m}        |
        | sub01(X1,X2) = {<s,t>} |
        |----------------------|
q-bf    | In(X) = {s}          |
        | sub11(Z) = {g}       |
        |----------------------|
        | In(X) = {g}          |
        | sub2(Y) = {k}        |
        | sub01(X1,X2) = {}    |
        | sub04(Y) = {}        |
p-bf    | Out(Y) = {k}         |
        ------------------------
                (a)

        ------------------------
        | In(X) = {a}          |
p-bf    | sub2(Y) = {m}        |
        | sub01(X1,X2) = {<s,t>} |
        |----------------------|
        | In(X) = {s}          |
q-bf    | sub11(Z) = {g}       |
        | Out(Y) = {k}         |
        ------------------------
                (b)

        ------------------------
        | In(X) = {a}          |
p-bf    | sub2(Y) = {m}        |
        | sub01(X1,X2) = {<s,t>} |
        | sub02(Y) = {k}       |
        |----------------------|
        | In(X) = {t}          |
q-bf    | sub11(Z) = {d}       |
        |----------------------|
        | In(X) = {d}          |
        | sub2(Y) = {k}        |
p-bf    | sub01(X1,X2) = {}    |
        | sub04(Y) = {k}       |
        | Out(Y) = {k}         |
        ------------------------
                (c)

        ------------------------
        | In(X) = {a}          |
        | sub2(Y) = {m}        |
p-bf    | sub01(X1,X2) = {<s,t>} |
        | sub02(Y) = {k}       |
        |----------------------|
        | In(X) = {t}          |
q-bf    | sub11(Z) = {d}       |
        | Out(Y) = {k}         |
        ------------------------
                (d)

        ------------------------
        | In(X) = {a}          |
        | sub2(Y) = {m}        |
        | sub01(X1,X2) ={<s,t>} |
p-bf    | sub02(Y) = {k}       |
        | sub03(Y) = {k}       |
        | sub04(Y) = {k}       |
        | Out(Y) = {m,k}       |
        ------------------------
                (e)
```

Figura 2.

La configurazione (a) della stack contiene, dall'alto, il record di attivazione per la procedura

p-bf con valore di input In(X)={a} e le variabili sub2(Y) e sub01(X1,X2), inizializzate dalle istruzioni 1 e 2 di p-bf (Figura 1). Il secondo record di attivazione in (a) corrisponde al richiamo della procedura q-bf, invocata all'istruzione 3 in p-bf, con valore di input In(X)={g}. Il terzo record di attivazione corrisponde alla procedura p-bf chiamata in q-bf, all'istruzione 2, con valore di input In(X)={s}. Quest'ultima chiamata di p-bf termina con Out(Y)={k}, che e' restituito a q-bf, cioe' al secondo record di attivazione della stack. Quando la procedura finisce, il suo record di attivazione e' rilasciato, e la stack assume la configurazione (b) della Figura 2. In modo analogo si potrebbero commentare le configurazioni (c), (d) ed (e).

La storia dell'allocazione e disallocazione dei records sulla stack puo' essere completamente rappresentata mediante un albero, detto albero di evoluzione della stack; in seguito, in questo lavoro, tale albero verra' utilizzato come strumento di supporto alle considerazioni sulle caratteristiche e sulle proprieta' dei metodi Counting ed Alexander.
L'albero dell'evoluzione della stack relativo all'esecuzione della procedura p-bf, con valore di input In(X) = {a}, e' rappresentato nella Figura 3.

Definizione: **l'albero della evoluzione della stack**, o semplicemente l'albero della stack, e' un albero ordinato che descrive l'esecuzione di una procedura in linguaggio convenzionale. I nodi corrispondono ai records di attivazione di una procedura a qualunque istante presenti sulla stack durante l'esecuzione, e contengono le variabili delle procedure ed i valori che esse hanno

- dopo che e' stata compiuta l'ultima operazione dell'esecuzione considerata, se si trovano in nodi posti su rami finiti;
- prima della chiamata alla procedura (associata ad un nodo figlio) che non termina, se si trovano in nodi su rami infiniti.

I nodi di livello j sono ordinati rispetto l'ordine con cui le procedure ad essi associate sono chiamate da quelle corrispondenti ai nodi genitori (di livello j-1).

Dato un nodo dell'albero della stack, si puo' parlare di procedura associata al nodo, di variabili di input e loro valori associati al nodo, di variabili di output e loro valori associati al nodo, di variabili locali al nodo, di predicato associato al nodo e viceversa.

Le configurazioni a stack come quelle presentate in Figura 2 sono cammini attraverso l'albero dalla radice a un nodo, con valori parziali per le variabili sull'albero della stack.

Albero dell'evoluzione della stack:

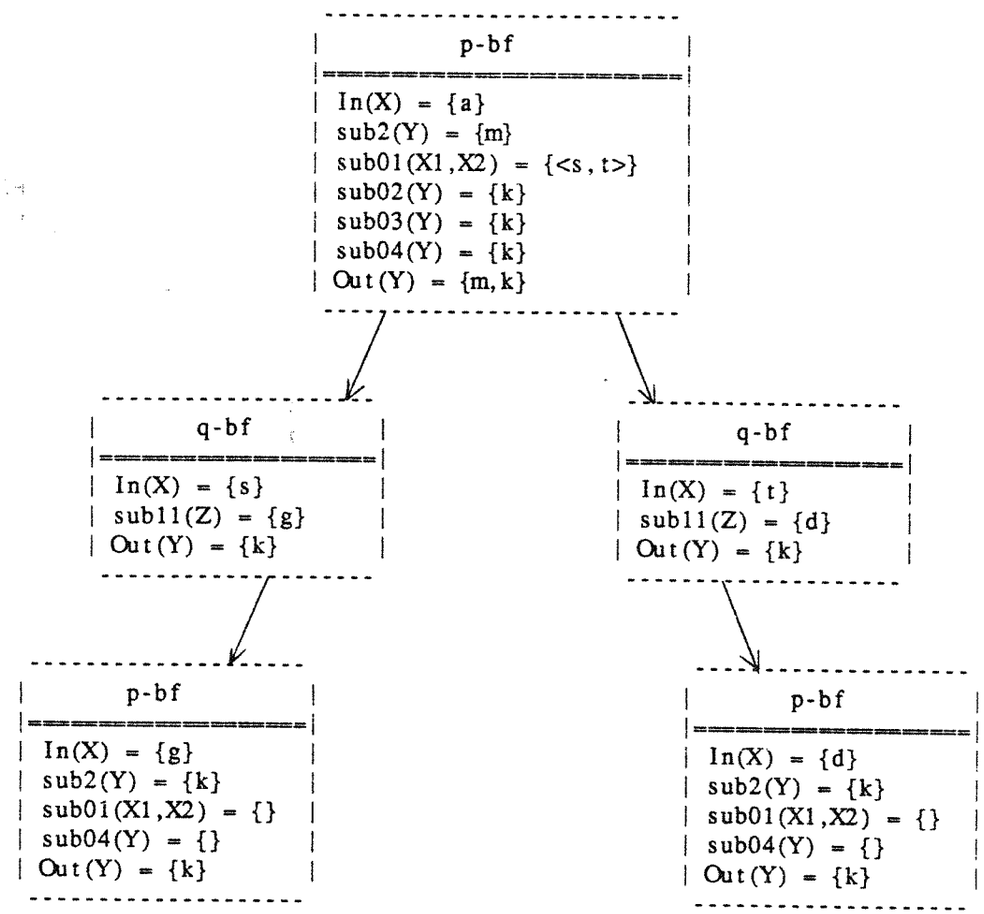


Figura 3

4. Etichettatura dell'albero.

Definizione: dato un programma logico LP, con m regole ricorsive, e dato un goal G, i nodi dell'albero della stack relativo alla valutazione del goal G su LP possono essere etichettati ciascuno con una tripla

<j, k, h>

definita nel seguente modo ricorsivo:
a) il nodo radice e' etichettato con <0, 0, 0>;

b) per ogni nodo n dell'albero etichettato con <j, k, h> e associato a un predicato q, ogni nodo figlo n', associato a un predicato del corpo di una regola che definisce q (e precisamente della i-esima regola di LP), e' etichettato con una tripla del tipo

$$<j+1,\ m*k+i,\ p_i*h+v>$$

dove
- i, come anticipato, e' l'indice della regola di LP che definisce q e nel cui corpo si trova il predicato associato ad n', considerando le regole numerate a partire da 0;
- p_i e' il numero di richiami ricorsivi presenti nell'i-esima regola ricorsiva di LP;
- v e' il numero d'ordine dell'occorrenza del predicato ricorsivo associato al nodo n', considerando i predicati ricorsivi nel corpo di una regola numerati a partire da 0.

Si puo' dare una spiegazione informale del significato di un'etichetta <j, k, h>:
- j indica il livello dell'albero in cui si trova il nodo;
- k individua uno o piu' sottoinsiemi di nodi dello stesso livello ciascuno dei quali risulta successore di uno stesso nodo n, e contribuisce alla produzione di una stessa sottorisposta;
- h indica il numero d'ordine, (partendo da sinistra), del nodo considerato, tra quelli che sono figli di uno stesso nodo n ed hanno j e k uguali.

L'etichettatura definita e' effettivamente una codifica dei nodi dell'albero [Port87].

5. Metodi di riscrittura di regole.

L'albero di evoluzione della stack puo' essere utilizzato per formulare una "rilettura" dei metodi Counting ed Alexander allo scopo di evidenziare alcune proprieta' di questi ultimi, ed in particolare le proprieta' di correttezza e completezza, terminazione ed efficienza.
La rilettura consiste nella definizione di una corrispondenza tra i nodi dell'albero della stack (relativo all'esecuzione delle procedure in linguaggio convenzionale associate ad un programma logico dato e ad un goal) e gli incrementi di relazione inseriti nella base di dati dall'esecuzione per saturazione del programma logico riscritto.

5.1. Rilettura del metodo Counting.

Il metodo Counting genera un insieme di clausole la cui esecuzione per saturazione produce come risultato delle relazioni del tipo cnt-p_i-s(J,K,H,X^*) e p_i-s(J,K,H,Y^*). I primi tre argomenti delle relazioni prodotte dal Counting costituiscono l'identificatore del nodo, in quanto corrispondono all' etichettatura dell'albero della stack definito nel paragrafo 3.

Esempio
Come esempio si applichi il metodo Counting all'insieme di clausole S, e sia S_1 l' insieme oggetto di clausole, tratto da [Sacc86].

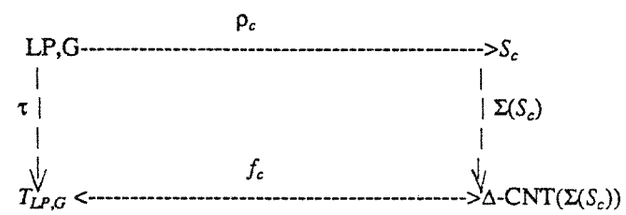
S_1:
(a1) cnt.p-bf(0, 0, 0, a).
(b1) cnt.p-bf(J, K, H, X), B1(X, x_1), B2(X, X_2) ---> cnt.q-bf(J+1, 2*K+0, 2*H+0, X_1).
(c1) cnt.p-bf(J, K, H, X), B1(X, X_1), B2(X, X_2) ---> cnt.q-bf(J+1, 2*K+0, 2*H+1, X_2).
(d1) cnt.q-bf(J, K, H, X), B4(X, Z) ---> cnt.p-bf(J+1, 2*K+1, 1*H+0, Z).
(e1) q-bf(J, K, H, Y), q-bf(J, K, H+1, Y), B3(Y, Z) ---> p-bf(J-1, (K-0)/2, H/2, Y).

(f1) p-bf(J, K, H, Y) ---> q-bf(J-1, (K-1)/2, H/1, Y).
(g1) cnt.p-bf(J, K, H, X), B5(X, Y) ---> p-bf(J, K, H, Y).
goal: p-bf(0, 0, 0, Y)?

La computazione guidata dall'antecedente delle clausole nell'insieme S_1 produce incrementi per le relazioni cnt.p-bf(J, K, H, X), cnt.q-bf(J, K, H, X), p-bf(J, K, H, Y) e q-bf(J, K, H, Y). Dopo la computazione, queste relazioni conterranno un insieme di tuple per ogni nodo <J, K, H> dell'albero della stack. Le tuple appartenenti a cnt.p-bf(J, K, H, X) e quelle di p-bf(J, K, H, Y) contengono rispettivamente i valori di input X e quelli di output Y della chiamata alla procedura p-bf corrispondente al nodo <J, K, H>. Invece le tuple che stanno in cnt.q-bf(J, K, H, X) e quelle di q-bf(J, K, H, Y) contengono rispettivamente i valori di input X e quelli di output Y della chiamata alla procedura q-bf corrispondente al nodo <J, K, H>.
In ogni classe di tuple {cnt-p_i-s(j, k, h, X^*)} generata dall'esecuzione per saturazione di un codice Counting (cioe' in ogni insieme di tuple aventi uguali i valori dei primi tre parametri) i valori assunti da X^* coincidono proprio con quelli delle variabili in ingresso al nodo <j, k, h> dell'albero della stack. La proprieta', piu' formalmente espressa dai seguenti teoremi e dai relativi corollari, si dimostra per induzione sul livello dei nodi dell'albero della stack.

Teorema 1. *Siano dati un insieme LP di clausole di Horn ed un goal G che soddisfino le condizioni di applicabilita' del Counting. L'algoritmo ρ_c proposto da tale metodo riscrive LP e G in un insieme di clausole S_c la cui valutazione bottom-up per saturazione $\Sigma(S_c)$ produce un risultato $\Delta(\Sigma(S_c))$ che puo' essere partizionato nei due sottoinsiemi Δ-$CNT(\Sigma(S_c))$ e Δ-$P(\Sigma(S_c))$, contenenti rispettivamente tutte le tuple di tipo cnt-p_i e tutte quelle di tipo p_i. Se l'albero di evoluzione della stack $T_{LP,G}$, prodotto da un'interpretazione τ su una macchina a stack degli originali LP e G, e' finito, allora esiste una funzione biiettiva f_c che ad ogni nodo di $T_{LP,G}$ associa una ed una sola classe di tuple di Δ-$CNT(\Sigma(S_c))$.*

Schematicamente questo enunciato puo' essere illustrato dal seguente diagramma.



Corollario 1.1. *Se l'albero $T_{LP,G}$ e' infinito, la funzione f_c: $T_{LP,G}$ ---> Δ-$CNT(\Sigma(S_c))$ e' una funzione totale e iniettiva.*

Nel paragrafo relativo alle proprieta' dei codici oggetto si fara' riferimento al precedente teorema per giustificare le considerazioni relative alla terminazione e all'efficienza del codice Counting.
Per la completezza e la correttezza si fara' invece uso del seguente teorema, nel cui enunciato i simboli LP, G, $T_{LP,G}$, Δ-P($\Sigma(S_c)$) assumono il significato definito dal precedente.

Teorema 2. *Dati un programma logico LP ed un goal G, se $T_{LP,G}$ e' finito ed i risultati restituiti da ogni nodo di $T_{LP,G}$ sono non vuoti, allora esiste una corrispondenza biunivoca g_c tra i nodi di $T_{LP,G}$ e le classi di tuple di Δ-$P(\Sigma(S_c))$*

Corollario 2.1. *Se l'albero $T_{LP,G}$ e' finito ed alcuni suoi nodi restituiscono risultato vuoto, la corrispondenza g_c e' una funzione biettiva parziale.*

La parzialita' della funzione e' dovuta al fatto che ai nodi di $T_{LP,G}$ che restituiscono risultato vuoto non corrisponde alcuna tupla di Δ-P($\Sigma(S_c)$).

Corollario 2.2. *Se l'albero $T_{LP,G}$ e' infinito la corrispondenza g_c e' una funzione parziale iniettiva.*

Per la dimostrazione dei teoremi e dei relativi corollari si puo' far riferimento a [Port87].

5.1.1. Semplificazioni.

Il metodo del Counting usa *una identificazione del nodo piu' semplice* se nell'insieme di clausole iniziale

a) il nome della procedura identifica un unico nodo al livello j, tra i nodi figli di un nodo a livello j-1 (cioe' nessuna clausola contiene un predicato ripetuto due o piu' volte) oppure
b) esiste soltanto una clausola ricorsiva oppure
c) entrambi.

In tali casi semplificati la precedente tripla per l'identificazione del nodo puo' essere ridotta a a) <J, K> oppure b) <J, H> oppure c) <J> rispettivamente.

5.2. Rilettura del metodo Alexander.

Secondo la formulazione presentata in [Rohm86], il metodo Alexander utilizza una valutazione da sinistra a destra dei predicati. Considerando, ad esempio, la prima clausola del programma logico (2) con goal p(a,Y), la trasmissione dell'informazione implica che la prima occorrenza del predicato q debba essere valutata con la sola prima variabile istanziata (q-bf), mentre la seconda con tutte e due istanziate (q-bb). Per analogia con i metodi precedenti si puo' scegliere di considerare le occorrenze del predicato q nella prima clausola in (2) come entrambe <bound,free>, anche quando si applica il metodo Alexander. Si ottiene cosi' l'insieme S_2.

S_2:

(a2) pb-p-bf(a).
(b2) pb-p-bf(X), B1(X, X_1) ---> pb-q-bf(X_1).
(c2) sol-q-bf(X_1, Y), pb-p-bf(X), B1(X, X_1), B2(X, X_2) ---> pb-q-bf(X_2).
(d2) sol-q-bf(X_2, Y), sol-q-bf(X_1, Y), pb-p-bf(X), B1(X, X_1), B2(X, X_2), B3(Y, Z) --->
---> sol-p-bf(X, Y).
(e2) pb-q-bf(X), B4(X, Z) ---> pb-p-bf(Z).
(f2) sol-p-bf(Z, Y), pb-q-bf(X), B4(X, Z) ---> sol-q-bf(X, Y).
(g2) pb-p-bf(X), B5(X, Y) ---> sol-p-bf(X, Y).

In Alexander le soluzioni al goal p(a, Y) sono tuple della relazione sol-p-bf(X, Y) aventi X=a.
La computazione guidata dall'antecedente delle clausole nell'insieme S_2 produce incrementi per le relazioni pb-p-bf(X), pb-q-bf(X), sol-p-bf(X, Y) e sol-q-bf(X, Y). Dopo la computazione queste relazioni contengono un insieme di tuple per ogni valore di input X di una chiamata a una procedura p-bf o q-bf. Le tuple in pb-p-bf(X) e le tuple in sol-p-bf(X, Y) contengono rispettivamente i valori di input X ed i valori di <input, output> <X, Y> di una chiamata a una procedura p-bf. Le tuple in pb-q-bf(X) e quelle in sol-q-bf(X, Y) contengono rispettivamente i valori di input X e quelli di <input, output> <X, Y> di una chiamata a una procedura q-bf.

Gli incrementi di relazione dell'insieme di regole ottenute applicando la riscrittura Alexander non codificano, oltre ai valori di input e di output, i nodi dell'albero della stack correlati alle chiamate di procedure. In generale, ad ogni nodo associato ad una procedura p dell'albero della stack corrispondono due insiemi di tuple, pb-p-s(X^*) e sol-p-s(X^*, Y^*), rispettivamente di tipo problema e di tipo soluzione, essendo X^* il vettore delle variabili di input relative a quel nodo ed Y^* il vettore delle variabili di output relative a quel nodo e all'input X^*. Le tuple delle relazioni pb-p-s e sol-p-s permettono di identificare la corrispondente procedura p-s ed i relativi argomenti, ma non consentono di associare ogni argomento ad un unico particolare nodo dell'albero della stack: una stessa tupla puo' infatti corrispondere a piu' nodi.
Cio' significa che

- per tracciare l'albero della stack con il metodo Alexander, e quindi per associare correttamente un risultato alla corrispondente chiamata di procedura, si devono considerare le relazioni esistenti tra i dati (piuttosto che trovarle immediatamente, come succede nel metodo Counting, attraverso parametri interi che identificano i nodi dell'albero): in questo caso cioe', per sfruttare i risultati relativi alla valutazione di un predicato ricorsivo p, si devono riconsiderare tutti i predicati che precedono la corrispondente istanziazione di p nel corpo della regola in esame;
- l'esecuzione guidata dai dati del codice riscritto secondo Alexander non puo' non terminare, in quanto inserisce nella base di dati un insieme di tuple che e' in ogni caso finito. Questa esecuzione produce un insieme di risultati che contiene, propriamente o impropriamente, quello prodotto in ambiente convenzionale dall'esecuzione delle corrispondenti procedure (e quindi quello fornito dalla valutazione guidata dal conseguente del programma logico dato); l'inclusione puo' essere propria se l'esecuzione delle procedure sulla macchina a stack entra in ciclo, mentre se quest'ultima termina i due insiemi necessariamente coincidono.

6. Proprieta' dei programmi oggetto.

In questo paragrafo si definiscono i concetti di equivalenza forte e di equivalenza debole rispettivamente tra l'esecuzione guidata dal conseguente del programma logico dato e quella guidata dall'antecedente dei codici riscritti secondo Counting e secondo Alexander.
Tali concetti sono definiti a partire dalle proprieta' di terminazione ed efficienza, completezza e correttezza dei risultati delle riscritture.

6.1. Terminazione ed efficienza.

La finitezza della base di Herbrand e' condizione sufficiente per la terminazione di un programma logico (che e' costituito da un numero finito di regole) eseguito secondo una strategia guidata dall'antecedente.
Se questa condizione viene a mancare, cioe' se gli argomenti dei predicati contengono simboli funzionali, e' possibile che ad ogni passo della computazione si riesca ad applicare una o piu' regole, ed a generare ogni volta fatti diversi dai precedenti.
Il problema della terminazione dell'esecuzione di un programma logico si pone ad esempio quando almeno una tra le relazioni ricorsive definite dallo stesso presenta un ciclo.
Se l'esecuzione e' guidata dal conseguente, l'esistenza di un ciclo consiste nel presentarsi, durante la computazione, di un sottogoal identico ad un altro gia' incontrato nella successione dei richiami ricorsivi annidati, per la valutazione del goal iniziale.

Negli esempi precedenti non si e' presa in considerazione la possibilita' del verificarsi di cicli, la cui presenza puo' comportare, nella valutazione del goal p(a,Y), sia la non terminazione dell'esecuzione guidata dal conseguente dell' insieme di clausole S , sia quella delle corrispondenti procedure in linguaggio convenzionale su una macchina a stack.

In caso di ciclo, la computazione sulla macchina relazionale dell'insieme di clausole generato dalla riscrittura Counting cicla esattamente allo stesso modo, a causa della simulazione esplicita della allocazione dei records di attivazione per mezzo dei parametri di livello della stack. Infatti l'uso del parametro che identifica il livello della chiamata procedurale (nelle tuple cnt.p-bf e cnt.q-bf, nell'esempio) non permette di riconoscere come uguali fatti che differiscono soltanto per il valore dello stesso.
In generale, in base al teorema 1 ed ai suoi corollari, se l'esecuzione guidata dal conseguente di un programma logico non termina, e quindi se il corrispondente albero della stack e' infinto, l'esecuzione guidata dall'antecedente della riscrittura Counting del programma stesso genera un insieme infinito di fatti, cioe' non raggiunge la condizione di terminazione.
Al contrario, la computazione della macchina relazionale relativa ad S_2, risultato dall'applicazione del metodo Alexander, si ferma correttamente dopo aver individuato tutte le risposte distinte al goal p(a, Y). Cio' accade poiche' una chiamata di procedura, che e' un sottoproblema da risolvere, e' conservata nella base di dati della macchina relazionale come pb-<nome del problema>(valore degli argomenti di input), ed e' quindi possibile riconoscere come identici sottoproblemi aventi uguali il nome ed i valori dei parametri di input, evitandone la rivalutazione.
Si osservi che il metodo Alexander paga questo "risparmio" con la necessita' di avere predicati ripetuti nel corpo delle clausole, alcune delle quali devono condividere i medesimi predicati gia' risolti. Infatti, mentre con il metodo Counting si riesce, utilizzando la codifica

costituita dai tre parametri J, K, H, ad associare direttamente ogni risposta con il problema a cui essa si riferisce, con la trasformazione Alexander e' necessario, per sfruttare un fatto di tipo soluzione, ripercorrere l'intero cammino che ha condotto alla formulazione del problema ad esso associato.
In riferimento all'esempio (2), il "join" tra i predicati pb-p-bf(X) e B1(X, X_1) compare in tre clausole di S_2.

L'esistenza dei cicli dipende strettamente dalla configurazione della base di dati, come evidenziato dall'esempio seguente.

Esempio.
Sia data la base di dati

r(a,b). r(b,c).
r(c,d). r(d,a).

Essendo la relazione r binaria, e' possibile darne una rappresentazione a grafo (secondo le indicazioni contenute in "An Amateur's Introduction to Recursive Query Processing Strategies" di Bancilhon e Ramakrishnan [Banc86]), in cui ogni tupla r(a_1,a_2) e' rappresentata da un arco labellato con r, avente come coda il nodo associato ad a_1, e come testa quello associato ad a_2.
La base di dati introdotta puo' quindi essere rappresentata dal seguente grafo, nel quale si omettono i labels degli archi poiche', essendo unica la relazione, non ci puo' essere ambiguita'.

```
a-------->b
^         |
|         v
d<--------c
```

Si consideri il programma sorgente:

p(X, Y) :- r(X, Y).
p(X, Y) :- r(X, Z), p(Z, T), r(T, Y).

con la query p(a, Y).
La trasformazione Counting, di cui si applica la versione semplificata poiche' il programma contiene un'unica regola ricorsiva che e' pure lineare, fornisce il seguente insieme di regoie:

cnt-p-bf(J, X), r(X, Y) ---> p-bf(J, Y).
cnt-p-bf(J, X), r(X, Z) ---> cnt-p-bf(J+1, Z).
p-bf(J, T), r(T, Y), J>0--->p-bf(J-1, Y).
cnt-p-bf (0, a).

Simulando la valutazione del goal su una macchina relazionale si puo' notare che al passo 4 viene generata la tupla cnt-p-bf(4,a), che rappresenta il proporsi di un problema in "a" al livello 4, problema gia' presente anche al livello 0 (come evidenziato dall'esistenza della tupla cnt-p-bf(0,a)), risolvendo il quale si e' giunti al passo 4. I fatti cnt-p-bf(0, a) e cnt-p-bf(4, a) sono ovviamente diversi, quindi, nel tentativo di saturare la base di dati, si

dovrebbe continuare l'esecuzione applicando ancora tutte le regole nel cui corpo compaiono predicati di cui cnt-p-bf(4, a) costituisce una istanziazione: cio' significa che il programma cicla all'infinito, in quanto vengono prodotti tutti i fatti del tipo cnt-p-bf(4*w, a) e, analogamente, cnt-p-bf(1+4*w, b), cnt-p-bf(2+4*w, c), cnt-p-bf(3+4*w, d), essendo w un numero naturale.

Il risultato dell'applicazione della riscrittura col metodo Alexander e' invece costituito dall'insieme di clausole;

pb-p-bf(X), r(X, Y) ---> sol-p-bf(X, Y).
pb-p-bf(Y), r(X, Z)---> pb-p-bf(Z).
pb-p-bf(X), r(X, Z), sol-p-bf(Z, T), r(T, Y)---> sol-p-bf(X, Y).
pb-p-bf(a).

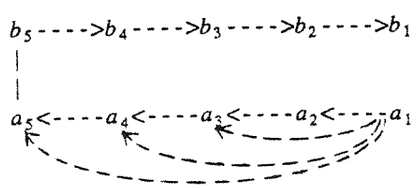
L' esecuzione del codice riscritto termina in quanto, dopo il passo 4, tutte le relazioni hanno incremento Δ vuoto. Cio' e' dovuto all'identificazione di fatti che, anche se prodotti in punti diversi della computazione, sono effettivamente uguali.

Anche l'esistenza di ripetizioni dipende strettamente dalla configurazione della base di dati su cui la relazione ricorsiva e' definita.
In particolare, assumendo la rappresentazione a grafo delle relazioni, una ripetizione puo' essere dovuta all' esistenza di almeno due cammini, di diversa lunghezza, da un nodo ad un altro [Banc86].
Nel valutare un goal p(a, Y) relativo ad una relazione virtuale p(X, Y), la capacita' di individuare le ripetizioni consiste nel riconoscere il fatto di aver gia' trovato una certa istanziazione, rappresentante un problema, e di aver gia' calcolato l'output per un certo dato di input per quel problema, in modo da poter evitare una perdita di tempo nel rifare calcoli gia' fatti.
Anche in questo caso, i programmi oggetto delle riscritture Counting e Alexander hanno, in fase di esecuzione, comportamenti diversi. A questo proposito si introduce un esempio di esecuzione con ripetizioni, costruito su un programma logico ed una base di dati tratti da [Banc85].

Esempio.
Si abbia la base di dati

q(a_5, b_5)
r(a_i, a_{i+1}) per i=1,...,4
r(a_1, a_i) per i=3,...,5
s(b_i, b_{i-1}) per i=2,...,5

la cui rappresentazione grafica e':



dove gli archi diretti da destra verso sinistra rappresentano tuple della relazione r, quelli diretti da sinistra verso destra sono relativi alla relazione s, mentre l'arco verticale si riferisce all'unica tupla di q.

Il grafo presenta, per la relazione r, coppie di nodi collegati da piu' cammini di lunghezze diverse: ci sono quattro differenti cammini da a_1 a a_5, aventi rispettivamente lunghezza 1, 2, 3 e 4; piu' cammini esistono anche tra i nodi a_1 ed a_4 e tra a_1 ed a_3.

Si consideri l'insieme di clausole:

p(X, Y) <--- q(X, Y).
p(X, Y) <--- r(X, X_1), p(X_1, Y_1), s(Y_1, Y).

con il goal p(a_1, Y).
L'applicazione del metodo Counting al programma logico dato, con le semplificazioni dell'identificatore del nodo che riducono la tripla <J, K, H> al solo <J>, fornisce l'insieme di regole:

cnt.p-bf(0, a_1).
cnt.p-bf(J, X), r(X, X_1) ---> cnt.p-bf(J+1, X_1).
cnt.p-bf(J, X), q(X, Y) ---> p-bf(J, Y).
p-bf(J, Y_1), s(Y_1, Y), J>0--->p-bf(J-1, Y).

La riscrittura Alexander e' invece:

pb-p-bf(a_1).
pb-p-bf(X), q(X, Y) ---> sol-p-bf(X, Y).
pb-p-bf(X), r(X, X_1) ---> pb-p-bf(X_1).
pb-p-bf(X), r(X, X_1), sol-p-bf(X_1, Y_1), s(Y_1, Y) ---> sol-p-bf(X, Y).

L'esecuzione del codice Counting comporta la generazione di quattro tuple, rispettivamente di livello 1,2,3 e 4, associate la problema p-bf con input "a_5", tre tuple relative allo stesso problema in "a_4" e cosi' via; con la riscrittura Alexander i sottoproblemi p-bf in "a_5", "a_4", "a_3" e "a_2" vengono invece memorizzati una volta sola, anche se si ripresentano nel corso della computazione.
Pensando di eseguire su una macchina a stack la procedura Pascal-like associata al programma logico dato, si constata che il numero di records di attivazione allocati sulla stack di esecuzione e' 4(+1), proprio come il numero dei "livelli della stack" contati dal metodo Counting.

Gli esempi proposti sono presentati e commentati in modo piu' completo in [Demo87] e in [Demo88].

6.2. Correttezza e completezza dei metodi.

La completezza del metodo Counting e' un'immediata conseguenza del teorema 2 e dei suoi corollari in base ai quali tutti i valori assegnati alle variabili di uscita relative al nodo radice dell'albero della stack compaiono anche in tuple del tipo p-s(0,0,0,Y^*) prodotte dall'esecuzione per saturazione del codice riscritto.
Lo stesso teorema ed il suo primo corollario dimostrano la correttezza della riscrittura Counting nel caso l'albero dell'evoluzione della stack sia finito, cioe' nel caso in cui l'esecuzione guidata dal conseguente del programma logico dato termini.

Si puo' pensare di dimostrare la correttezza del Counting relativa al caso di programmi logici non terminanti riconducendo il comportamento della macchina a stack a quello della macchina relazionale, nel modo descritto nelle considerazioni sul secondo corollario del teorema 2.

Per quanto riguarda il metodo Alexander, si omettono le dimostrazioni sulla correttezza e completezza che gia' sono state fornite dagli autori del metodo stesso.

6.3. Definizione delle equivalenze.

La rilettura dei metodi Counting ed Alexander in ambiente di programmazione convenzionale e la definizione delle funzioni (rispettivamente biettiva e suriettiva) tra l'esecuzione forward di un programma logico riscritto e quella delle procedure in linguaggio imperativo ad esso corrispondenti permettono di dimostrare che:

a) tra l'esecuzione guidata dal conseguente di un programma logico LP dato e quella guidata dall'antecedente del corrispondente programma oggetto Counting S_c si definisce una *equivalenza forte*:
- se l'esecuzione guidata dal conseguente di LP termina, produce gli stessi risultati che si ottengono con l'esecuzione guidata dall'antecedente di S_c;
- per le due esecuzioni coincidono le condizioni di terminazione;
- i motivi di inefficienza, nel caso di ripetizioni, sono gli stessi;

b) tra l'esecuzione guidata dal conseguente di un programma logico LP dato e quella guidata dall'antecedente del corrispondente programma oggetto Alexander S_a si definisce una *equivalenza semplice*:
- se l'esecuzione guidata dal conseguente di LP termina, produce gli stessi risultati che si ottengono con l'esecuzione guidata dall'antecedente di S_a;
- l'esecuzione guidata dall'antecedente di S_a termina in ogni caso. Nei casi in cui l'esecuzione guidata dal conseguente di LP non termini, le due esecuzioni assumono una semantica diversa;
- l'esecuzione del codice oggetto Alexander non presenta motivi di inefficienza dovuti a ripetizioni.

7. Conclusioni

Oggetto di questo lavoro e' la rilettura dei metodi Counting ed Alexander in un ambiente di programmazione tradizionale.
Si e' innanzi tutto proposta un'interpretazione procedurale dei programmi logici, associando ad ogni predicato in essi definito (e ad ogni istanziazione delle sue variabili) una procedura in linguaggio convenzionale. Per le procedure cosi' definite si e' poi pensato ad una esecuzione sulla macchina a stack, della quale sono noti il comportamento e le proprieta', e si e' dimostrato che le riscritture Counting ed Alexander di un programma logico sono equivalenti, rispettivamente in modo forte e in modo semplice, all'esecuzione delle procedure in linguaggio convenzionale ad esso corrispondenti.
In tal modo si e' quindi potuto concludere che il codice generato dalla trasformazione Counting eredita le proprieta' e le caratteristiche dell' esecuzione sulla macchina a stack ad esso corrispondente, e tra queste le condizioni di terminazione ed i motivi di inefficienza, mentre l'esecuzione del codice generato dalla trasformazione Alexander termina in ogni caso, e non presenta motivi di inefficienza dovuti all'esistenza di ripetizioni.
Si e' inoltre dimostrato che tutti i risultati che si otterrebbero con l'esecuzione (secondo strategie guidate dal conseguente) del programma logico dato, sono forniti anche dall'esecuzione (secondo strategie guidate dall' antecedente) dei codici riscritti; queste ultime possono produrre piu' soluzioni, qualora l'esecuzione del programma logico dato entri in ciclo.

BIBLIOGRAFIA

[Banc86] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman, Magic sets and other ways to implement logic programs, MCC Rapporto Tecnico DB-121-85, 1985, ed in Atti conf. PODS 86, 1986.

[Banc86b] F.Bancilhon, R.Ramakrishnan, An amateur's introduction to recursive query processing strategies, Atti conf. SIGMOD 86, Washington,1986.

[Demo86] B. Demo, Recursive versus iterative schemes for least fix point computation in logic database, Rapporto interno Dipartimento di Informatica, Universita' di Torino, Aprile 1986 (apparso in Atti conf. COMPDEC 87, Los Angeles, 1987).

[Demo87] B. Demo, M. Porta, M.L. Sapino, Rewriting Rule Methods, Atti 5ª Conferenza sull'Intelligenza Artificiale, North Holland, Bratislava, Ottobre 1987.

[Demo88] B. Demo, M. Porta, M.L. Sapino, Efficiency and Completeness properties of the Counting Method, apparira' in Computers and Artificial Intelligence Journal, Bratislava, 1988.

[VanE76] M. H. Van Emden, R. A. Kowalski, The semantics of Predicate Logic as a Programming Language, JACM, Vol. 23, No. 4, 1976

[Forg81] C. L. Forgy, OPS5 User's Manual, CMU-CS-81-135 Report, Luglio 1981.

[Hens81] L. J. Henschen, S. A. Naqvi, On Compiling Queries in Recursive First-Order Databases, JACM Vol. 31, N. 1, 1981, pp. 372-392.

[Port87] M. Porta, M.L. Sapino, Metodi di riscrittura di clausole di Horn per la valutazione di predicati ricorsivi, Tesi di laurea, Universita' degli Studi di Torino, 1987.

[Rohm85] J.Rohmer, R. Lescoeur, La methode d'Alexandre, Rapporto Interno IA 4501 Centro di Ricerca BULL, Marzo 1985.

[Rohm86] J.Rohmer, R. Lescoeur, J. M. Kerisit, The Alexander method - A technique for the processing of recursive axioms in deductive databases, in Journal of New Generation Computing, N. 4, 1986.

[Sacc86] D. Sacca', C. Zaniolo, The generalized Counting Method for Recursive Logic Queries, Atti conf. ICDT '86, Roma, 1986.