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:

• propagazione delle strutture dati in avanti ed indietro.
• macro-espansione di procedure.
• valutazione di predicati built-in.

La stretta relazione tra valutazione parziale e interpretazione suggerisce come implementazione di tale tecnica un'estensione dell'interprete stesso.

La maggior efficienza del programma generato dalla valutazione parziale rispetto a quello originale è dovuta essenzialmente ad un numero minore di chiamate di procedura ed ad una specializzazione rispetto ai valori di ingresso noti.

Le proprietà fondamentali che un algoritmo di valutazione parziale deve soddisfare sono le seguenti:

• *terminazione*
• generazione di un programma *equivalente semanticamente* a quello originale rispetto ad un'istanziazione dei parametri di ingresso, e *più efficiente*.

2. La Valutazione Parziale in Programmazione Logica

Il principio della valutazione parziale si applica in modo naturale ai programmi logici, in quanto i meccanismi di unificazione e risoluzione presenti nell'interprete standard permettono rispettivamente la propagazione delle strutture dati in avanti e indietro e la riduzione del numero di chiamate di procedura.

Inoltre i valori di ingresso non devono essere necessariamente costanti, ma in generale saranno strutture dati parzialmente istanziate, cioè termini contenenti variabili logiche.

Allo stato attuale esistono in letteratura diversi lavori relativi all'applicazione della valutazione parziale a programmi logici:

• definizione di un valutatore parziale per Prolog [Komorowski-81] attraverso l'estensione di una macchina astratta Prolog specificata nel linguaggio Meta-IV.
• definizione di "unfold-fold" [Tamaki-84], tecnica di trasformazione di programmi analoga alla valutazione parziale.
• definizione di un valutatore parziale per Full Prolog, con interessante applicazione all'ottimizzazione delle interrogazioni a basi di dati [Venken-84].
• definizione di un valutatore parziale basato sulla SLD Resolution [Ghelfo-87], realizzato per l'applicazione a metaprogrammi in un linguaggio logico con teorie multiple [Levi-87].

L' algoritmo di valutazione parziale proposto in questo articolo segue la definizione suggerita in [Ghelfo-87].

In programmazione logica infatti la valutazione parziale può essere vista come una procedura che dati un programma P ed un goal G parzialmente istanziato genera un

nuovo programma P'che ha lo stesso comportamento di P rispetto a G ed a ogni goal ottenuto istanziando ulteriormente G.

Tale procedura si basa essenzialmente sulla SLD Resolution [Lloyd-84] e può essere facilmente definita come un'estensione del metainterprete puro.

Considerando un programma P in HCL (Horn Clauses Language) ed un goal G, la valutazione parziale di P rispetto a G è un programma P' composto dalle seguenti clausole:

$$[G]\lambda_1 :\text{-} g_1$$
$$\cdot$$
$$\cdot$$
$$[G]\lambda_n :\text{-} g_n$$

dove:

• ogni clausola è relativa ad un cammino i appartenente all'albero SLD di P con radice G [i=1..n]
• λ_i è la sostituzione associata al cammino i.
• g_i è l'etichetta del nodo foglia del cammino i.

Se il cammino i-esimo è finito, g_i è uguale a [] (clausola vuota) o no, a seconda che la derivazione i-esima abbia avuto successo o fallimento. In questo ultimo caso g_i sarà il sottogoal contenente il letterale che ha determinato il fallimento. Le clausole relative a derivazioni con fallimento vengono ugualmente inserite in P', in quanto il programma P potrebbe essere specificato solo parzialmente al momento della valutazione parziale (*definizione incrementale di P*), e tali fallimenti possono essere causati dalla mancanza di definizioni che in seguito potranno essere disponibili.

Se il cammino i-esimo è infinito si pone il problema di individuare un sottogoal g_i, in maniera che l'algoritmo di valutazione parziale termini e che P' sia equivalente semanticamente a P per le istanziazioni parziali del goal G.

3. Valutazione Parziale di Programmi HCL ricorsivi: problemi.

Nel caso di programmi logici ricorsivi garantire le proprietà di terminazione e di equivalenza implica il riconoscimento di rami infiniti di un albero SLD e la selezione in essi di un opportuno sottogoal in maniera che sia possibile definire un programma equivalente e più efficiente di quello originale.

Il problema di definire una strategia di terminazione, ossia di riconoscere rami infiniti, non è decidibile in generale.

Inoltre il sottogoal corrispondente alla derivazione che viene bloccata deve permettere la definizione di un programma semanticamente equivalente a quello originale.

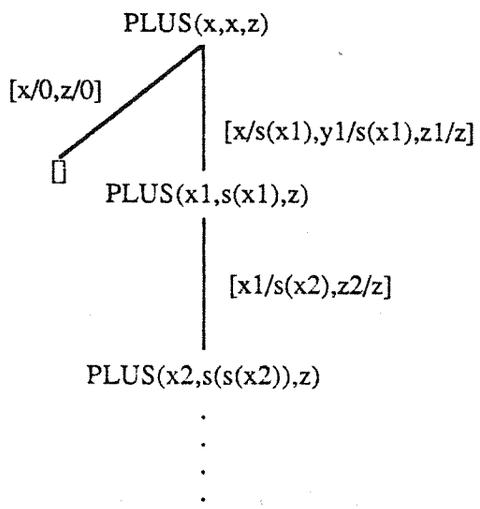
Le strategie di terminazione proposte in questo contesto sono quelle che interrompono una derivazione qualora vengano identificati letterali che sono istanze, varianti, o unificabili con letterali già precedentemente incontrati nello stesso cammino.

Anche in casi comuni però tali strategie non sono corrette, in quanto non garantiscono nè che l'algoritmo termini, nè l'equivalenza tra il programma generato e il sorgente, come mostrato dai seguenti esempi:

Sia P il programma composto dalle seguenti clausole:

PLUS (s(x),y,s(z)) :- PLUS (x,y,z)
PLUS (0,y,y) :-

La valutazione parziale di P rispetto al goal G= PLUS(x,x,z) genera il seguente albero SLD:



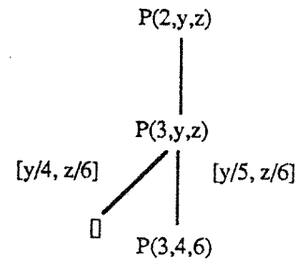
Adottando come strategia di terminazione l'individuazione di un letterale istanza di un antenato, la valutazione parziale di P rispetto a G non termina in quanto tale condizione

non viene mai verificata.

Un ulteriore esempio di non correttezza di tale strategia è il seguente:
Consideriamo il programma T, composto da:

P(2,y,z) :- P(3,y,z)
P(3,5,6) :- P(3,4,6)
P(3,4,6):-

La valutazione parziale di T rispetto al goal P(2,y,z) genera il seguente albero SLD:



Poichè il sottogoal P(3,4,6) è istanza di P(3,y,z) la derivazione viene interrotta, ed il programma T' generato dalla valutazione parziale é il seguente:

P(2,4,6) :-
P(2,5,6) :- P(3,4,6)

E' possibile notare che T e T' non sono equivalenti rispetto ad una qualsiasi istanziazione di G: infatti P(2,5,6) è conseguenza logica di T, ma non di T'.

L'approccio ai precedenti problemi, proposto in questo articolo, consiste nell'individuare alcune classi di programmi ed in base alle loro caratteristiche definire un algoritmo di valutazione parziale con una strategia di terminazione ad hoc.

In seguito verranno introdotte queste classi, verrà definito l'algoritmo di valutazione parziale, e verranno presentate le proprietà di terminazione e di generazione di programmi equivalenti e più efficienti.

4. Programmi con *Dominio Finito (DF).*

La base di Herbrand dei programmi con *Dominio Finito* (DF), cioé l'insieme di tutti gli atomi ground che possono essere formati componendo i predicati con i termini dell'universo di Herbrand, é finita. Questo é dovuto al fatto che nella definizione di programmi DF non compaiono simboli di funzione o costruttori. Questa classe comprende i programmi di tipo *Data-base*, cioé i programmi che rappresentano in pratica la parte deduttiva di Data-base.

5. Programmi *Primitivi Ricorsivi (PR).*

I programmi HCL *Primitivi Ricorsivi* (PR) sono i programmi trasformabili in funzioni primitive ricorsive. Più precisamente le funzioni computate da tali programmi includono tutte le funzioni Primitive Ricorsive.

Le clausole componenti tali programmi saranno quindi di tipo unit, o definite per composizione o ricorsione, analogamente a quanto accade per le funzioni primitive ricorsive.

Nei prossimi paragrafi forniamo una traccia di tale caratterizzazione dei programmi PR, definita formalmente in [Aquilano-85], ponendo l'accento sulle principali caratteristiche che sono state sfruttate per dimostrare le proprietà dell'algoritmo di valutazione parziale.

5.1. Definizione di selezione di I/O

La selezione di ingresso/uscita é una partizione degli argomenti di un predicato in due sottoinsiemi: *input* e *output*.

Intuitivamente gli argomenti di output vengono "calcolati" attraverso una refutazione SLD in base a quelli di input.

Definizione 1: Una selezione di I/O di $R(t_1,...,t_n)$ é definita nel seguente modo:

$<>R$ se $n=0$

$<S_1,....,S_n>R$ se $n>0$, dove $S_j \in \{in , out\}$ e deve esistere almeno un S_j' per cui $S_j'=in$.

Definizione 2: Consideriamo un programma P che definisce *m* relazioni $R_1,.....,R_m$. Per ogni relazione n-aria R_j sia rs_j l'insieme di tutte le selezioni di I/O per R_j. Allora una selezione di I/O per P é costituita da un elemento $rs \in RS$ dove $RS=rs_1 \times ... \times rs_m$.

Non tutte le possibili divisioni della tupla degli argomenti consentono di vedere la proiezione di un predicato come una funzione; attraverso *le regole per l'ammissibilità delle selezioni di I/O per singole clausole* è possibile individuare una selezione di I/O per cui la sottotupla di input può essere vista come l'insieme degli argomenti di una funzione e quella di output come il risultato dell'applicazione della funzione a tale input.

5.2. Regole di ammissibilità della selezione di I/O per singole clausole.

Le condizioni di ammissibilità fanno riferimento alle seguenti definizioni.
Siano $R(t_1,...,t_n)$ un generico atomo e rs una selezione di I/O. Allora si definiscono:

- $\mathbf{IN}_{rs}(R(t_1,...,t_n))$ l'insieme di tutte le variabili che occorrono nei t_j tale che S_j=in.
- $\mathbf{OUT}_{rs}(R(t_1,...,t_n))$ l'insieme di tutte le variabili che occorrono nei t_j tale che S_j=out.
- $\mathbf{MIN}_{rs}(R(t_1,...,t_n))$ il multi-insieme di tutte le variabili che occorrono nei t_j tale che S_j=in.
- $\mathbf{MOUT}_{rs}(R(t_1,...,t_n))$ il multi-insieme di tutte le variabili che occorrono nei t_j tale che S_j=out.
- $\mathbf{INPUT}_{rs}(R(t_1,....,t_n))$ l'insieme di tutti i termini t_j tale che s_j=in, con j diverso dalla posizione di ricorsione nel caso che il predicato R rappresenti una funzione primitiva ricorsiva definita per ricorsione.
- $\mathbf{OUTPUT}_{rs}(R(t_1,....,t_n))$ l'insieme di tutti i termini t_j tale che s_j=out.

Ammissibilità per clausole unit: Sia C = A:- una clausola unit. Allora una *selezione di I/O*, rs, per C *é ammissibile* se:

- $\mathbf{MIN}_{rs}(A)=\mathbf{IN}_{rs}(A)$

(la stessa variabile non compare in due termini di input *-linearità a sinistra-*)

- $\mathbf{IN}_{rs}(A) \supseteq \mathbf{OUT}_{rs}(A)$

(tutte le variabili di output devono comparire in input)

- $\mathbf{VAR} \supseteq \mathbf{INPUT}_{rs}(A)$

dove VAR è l'insieme di tutte le variabili. Questa regola ci assicura che la funzione, ottenuta con la trasformazione del programma, sia totale.

Ammissibilità per clausole non unit: Sia C= A:- $B_1,\ldots,B_n$ una clausola. Allora una *selezione di I/O*, rs, per C *é ammissibile* se:

- $MIN_{rs}(A)=IN_{rs}(A)$
  (linearità a sinistra)

- $IN_{rs}(A) \cap OUT_{rs}(B_j)=\emptyset$, per $j=1,\ldots,n$

- $\otimes_j MOUT_{rs}(B_j)=\cup_j OUT_{rs}(B_j)$, per $j=1,\ldots,n$
  dove $\otimes_j$ indica l'unione dei multi-insiemi e $\cup_j$ l'unione di insiemi (Ciascuna variabile di output deve essere calcolata una sola volta).

- $(\cup_j OUT_{rs}(B_j)) \cup IN_{rs}(A) \supseteq OUT_{rs}(A)$
  (Gli output della testa della clausola devono essere o input della testa stessa o calcolati dal corpo della clausola)

- $((\cup_j OUT_{rs}(B_j))-OUT_{rs}(B_i)) \cup IN_{rs}(A) \supseteq IN_{rs}(B_i)$
  Ogni input di un atomo, appartente al corpo della clausola, deve essere o calcolato da un atomo diverso del corpo o un input della testa . La terza e la quinta regola assicurano che non esistano variabili che compaiono sia come input sia come output in un atomo del corpo.

- **Per ogni $B_i = R(t_1,\ldots,t_n)$ e la corrispondente selezione di I/O, $<s_1,\ldots,s_n>R$, se s_j=out allora t_j é una variabile, per ogni $j=1,\ldots,n$.**
  Tutti gli argomenti di output nel corpo della clausola devono essere variabili.

- **Ogni B_i appartiene a $\cup_j REACH_{rs}[j](B_i)$**
  dove $REACH_{rs}[1](B_i)=\{B_j \mid \exists x : x\in OUT_{rs} \text{ e } x\in IN_{rs}(B_j)\}$ e
  $REACH_{rs}[k+1](B_i)=\{B_j \mid \exists B_s\in REACH_{rs}[k](B_i) \mid \exists x : x\in OUT_{rs}(B_s) \text{ e } x\in IN_{rs}(B_j)\}$

- $VAR \supseteq INPUT_{rs}(A)$
  regola analoga a quella relativa alla clausola unit.

5.3. Algoritmo di decisione per la classe dei *programmi primitivi ricorsivi*

L'algoritmo di decisione della classe dei programmi HCL PR é basato sui seguenti passi:

1) controllo dell'esistenza di una selezione di I/O ammissibile per un programma (il controllo sull'esistenza dell'ammissibilità della selezione di I/O relativa ad un programma P garantisce la trasformabilità di tale programma in notazione funzionale)

2) trasformazione del programma in notazione funzionale mediante tale selezione.

3) controllo della primitiva ricorsività della funzione ottenuta mediante il secondo passo.

Il seguente esempio mostra un programma e la funzione primitiva ricorsiva corrispondente:

Consideriamo il seguente programma T:

S(x,s(x)):-
P(0,x,x):-
P(s(x),y,s(z)):- S(w,z),P(x,y,w)

Una possibile partizione degli argomenti di input é (<in in out> P <in out>S) per T: in questo modo si ottengono le seguenti funzioni:

S(x)=s(x)
P'(0,x)=x
P'(s(x),y)=(S(P(x,y))

6. Descrizione dell'algoritmo di valutazione parziale

In questo paragrafo viene proposto un algoritmo di valutazione parziale che per i programmi appartenenti alle classi PR o DF soddisfa le proprietà introdotte nel paragrafo 1.

Tale algoritmo é essenzialmente basato sulla resolution SLD; nel caso in cui l'albero SLD relativo ad un programma P ed ad un goal G sia finito, lo spazio di ricerca della

resolution SLD e dell'algoritmo di valutazione parziale del programma P rispetto al goal G coincidono.

In caso contrario l'algoritmo proposto si basa sulla costruzione di sottoalberi SLD finiti opportunamente selezionati attraverso una condizione di riconoscimento di rami infiniti.

Premettiamo alcune definizioni a cui verrà fatto riferimento nella descrizione dell'algoritmo.

Definizione 4: relazione di antenato tra due sottoalberi

Siano S e T due sottoalberi aventi come radici G_s e G_t. Allora S é antenato di T sse esiste un sottogoal gi foglia di S in cui compare G_t (inoltre diremo che g_i é il sottogoal *padre di* G_t).

La relazione di antenato tra sottoalberi soddisfa la *proprietà transitiva.*

Definizione 5: relazione di antenato tra due letterali

Siano A e B due letterali appartenenti a due sottogoal g_k e g_s di due sottoalberi C_i e C_j, con radice R_i e R_j rispettivamente. Allora A é antenato di B sse:

$C_i = C_j$ Sia $R(g_k)$=A e sia γ l'mgu tra A e la clausola c_i.

i) B appartiene a $t(c_i)\gamma$, dove t identifica la parte destra di una clausola.

Oppure:

ii)$B=B'\alpha_1....\alpha_n$, dove B' appartiene a $t(c_i)\gamma$, $g_{k+1}=D_1,.,B',..,D_n$ e α_i é l'mgu i-esimo derivato dall'espansione di $D_1,....,D_n$.

$C_i \neq C_j$ R_j radice di Cj é antenato di B ed esiste una sequenza finita di sottoalberi $S_1,..,S_n$ con radici $R_1,....,R_n$ tali che:

- S_n é antenato di Cj e R_n é antenato di R_j in S_n.
- S_k é antenato di S_{k-1} e R_k é antenato di R_{k-1} in S_{k-1}[k=1..n-1]
- C_i é antenato di S_1 e A é antenato di R_1 in C_i.

La relazione di antenato tra letterali soddisfa la *proprietà transitiva.*

Definizione 6: condizione Θ

Un letterale A soddisfa la condizione Θ sse soddisfa la condizione Θ'oppure la condizione Θ":

• A soddisfa la condizione Θ' sse esistono almeno due indici **i** e **j** t.c. $s_i=s_j$=in, **i** é l'indice della posizione di ricorsione del predicato A e $A[i]=costr^n(x)$ e $A[j]=costr^{m(x)}$ con n,m>0, dove A[i] è l'i-esimo argomento del predicato A.

•A soddisfa la Θ" sse esistono nella derivazione di A due sottoalberi C_r e C_s che contengono due sottogoal, g_r e g_s, in cui compaiono due letterali A' e A" tali che A' è variante ed antenato di A".

Osservazione

Nel paragrafo 7 verrà mostrato che la condizione Θ risulta essere una condizione di terminazione necessaria e sufficiente per i programmi DF o PR.

Nella descrizione dell'algoritmo useremo le seguenti notazioni:

– R indica la regola di computazione usata, che nel nostro caso può essere la leftmost o la rightmost.

– $R(g_i)$ indica il letterale selezionato da R nel sottogoal g_i.

Definizione 7: regola di selezione dei letterali di un generico sottogoal

R' seleziona i letterali di un generico sottogoal g_i non precedentemente bloccati o non corrispodenti a fallimento nel seguente modo:

$R'(g_i)=G$ dove G é il letterale più a destra o più a sinistra in g_i (a seconda di R) che non è "bloccato" e non ha dato luogo ad un fallimento (la definizione di letterale "bloccato" sarà descritta nella seguente definizione).

Definizione 8: regola di "computazione" R* usata nell'algoritmo di valutazione parziale

La regola di computazione R* é ottenuta modificando la regola di computazione R nel seguente modo:

Sia g_i un generico sottogoal di un sottoalbero C_i con radice B_i.

1) $R'(g_i)$ viene "bloccato", ossia la sua derivazione SLD viene interrotta, sse soddisfa la Θ oppure è variante di un letterale che è gia stato bloccato.

2) $R'(g_i)$ viene derivato nel sottoalbero C_i sse è verificata una delle seguenti

condizioni:

i) $R'(g_i)$ non soddisfa la Θ e ha lo stesso "nome" (predicato) di R_i.

ii) $R'(g_i)$ non soddisfa la Θ e la clausola con cui è unificabile non costituisce una definizione direttamente o indirettamente ricorsiva per $R'(g_i)$.

3) Se $R'(g_i)$ .non è unificabile con nessuna clausola, cioè corrisponde ad un fallimento finito, vengono esaminati ugualmente gli altri letterali appartenenti a g_i.

4) altrimenti viene costruito un nuovo albero SLD di radice $R'(g_i)$ sse $R'(g_i)$ é unificabile con una clausola che costituisce una definizione direttamente o indirettamente ricorsiva per $R'(g_i)$.

5) se g_i contiene tutti i letterali che sono stati "bloccati", o corrispondono a fallimenti, allora tale sottogoal non viene ulteriormente espanso.

6) Se la derivazione dell' albero con radice B_i secondo la regola R* dà luogo ad un sottoalbero finito C_i, cioè in cui tutte le foglie sono uguali a [] oppure sono formate da letterali corrispondenti a fallimenti, il sottoalbero C_i viene"attaccato" al sottoalbero padre C_{i-1} (se esiste), ossia B_i viene derivato in C_{i-1}.

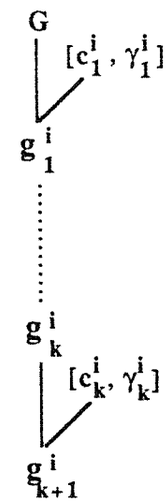
7) Se invece le foglie del sottoalbero ottenuto dalla derivazione di B_i contengono letterali "bloccati" allora anche B_i viene bloccato nel sottogoal in cui compare nel sottoalbero padre.

6.1 Algoritmo di valutazione parziale

Siano P un programma e G un goal. Allora l'algoritmo di valutazione parziale di $P \cup \{G\}$ corrisponde alla derivazione SLD di G secondo P mediante la regola di "computazione" R*. L'algoritmo di valutazione parziale di $P \cup \{G\}$ termina quando le foglie del sottoalbero di radice G sono uguali a [], oppure contengono letterali bloccati o corrispondenti a fallimenti finiti.

6.2 Costruzione del programma P' risultante dalla valutazione parziale di $P \cup \{G\}$

Per ogni ramo del sottoalbero relativo alla valutazione parziale di $P \cup \{G\}$ viene ricavata una clausola nel seguente modo: consideriamo un generico ramo del sottoalbero con radice G:



Se il sottogoal foglia del ramo i-esimo è uguale a [] allora P' contiene la seguente clausola:

$[G](\gamma_1 \ldots \gamma_k)^i$:-

Se invece il sottogoal foglia del ramo i-esimo è formato dai seguenti letterali:

$B^1(\ldots),\ldots\ldots\ldots B^n(\ldots)$

dove ogni $B^i(\ldots)$, per il passo1 e 5 della regola R*, è stato bloccato o corrisponde ad un fallimento finito, allora P' contiene la seguente clausola:

$[G](\gamma_1 \ldots \gamma_k)^i$:- $B1(\ldots),\ldots\ldots\ldots Bn(\ldots)$

Inoltre in P' saranno presenti le clausole derivate dai sottoalberi generati dall'algoritmo e aventi come radice ogni $B^i(\ldots)$ che non corrisponde ad un fallimento finito.

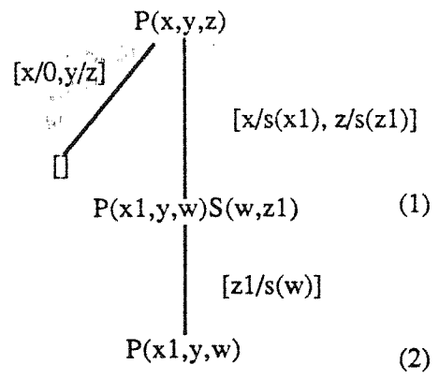
6.3 Un esempio di applicazione dell'algoritmo.

Si consideri il seguente programma T appartenente alla classe PR:

S(x, s(x)) :-

P(0,z,z) :-
P(s(x),y,s(z)) :- P(x,y,w), S(w,z)

Valutiamo parzialmente il programma T rispetto al goal P(x,y,z) secondo l'algoritmo proposto con regola di computazione R leftmost.



Al passo (1) il letterale P(x1,y,w) viene bloccato poiché soddisfa la condizione Θ, in quanto variante di un suo antenato (P(x,y,z)). Si considera quindi il letterale S(w,z1) che viene derivato direttamente nel sottoalbero. Al passo (2), essendo P(x1,y,w) variante di un letterale già "bloccato", l'algoritmo termina producendo il seguente programma T':

P(s(x), y, s(s(z))) :- P(x,y,z)
P(0,y,y) :-

7 Proprietà dell'algoritmo di Valutazione Parziale

In questo paragrafo vengono enunciate le principali proprietà dell'algoritmo di valutazione parziale, la cui dimostrazione formale é riportata in [Fanti-86].

Terminazione

La valutazione parziale di un programma appartenente alle classi PR o DF mediante l'algoritmo proposto termina sempre. Tale risultato è dovuto essenzialmente al fatto che le condizioni date per il riconoscimento dei cammini SLD infiniti risultano essere necessarie e sufficienti per tali classi di programmi e quindi l'algoritmo di valutazione parziale interrompe effettivamente tutte e sole le derivazioni infinite.

Equivalenza semantica

Dati un programma P, appartenente alla classe dei programmi PR o DF, ed un goal G, l'algoritmo di valutazione parziale proposto genera un programma P' che è semanticamente equivalente a P rispetto a G ed a una sua qualsiasi istanziazione.

Tale equivalenza semantica è dimostrabile in termini di semantica operazionale: in particolare è possibile dimostrare che le derivazioni di successo di P e P' rispetto ad una qualsiasi istanziazione di G hanno associate sostituzioni identiche a meno di varianza. In altri termini $\forall\gamma$ σ è r.c.a di $P\cup\{G\gamma\}$ sse σ è r.c.a di $P'\cup\{G\gamma\}$.

Inoltre per il teorema di correttezza della Resolution SLD è possibile osservare che P e P' hanno la stessa semantica dichiarativa, ossia $\forall\gamma$ $P\cup\{G\gamma\}$ è insoddisfacibile sse $P'\cup\{G\gamma\}$ è insoddisfacibile.

Efficienza

Il programma P' risultante dalla valutazione parziale di $P\cup\{G\}$ è effettivamente più efficiente di P: infatti per ogni goal G', istanziazione di G, lo spazio di ricerca di $P'\cup\{G'\}$ presenta le seguenti caratteristiche:

- riduzione della lunghezza (passi) delle derivazioni sia di successo che di fallimento.
- riduzione del numero delle derivazione di fallimento finito.
- riduzione del numero dei fallimenti infiniti, se tali fallimenti sono dovuti all'uso di una regola di computazione non fair.

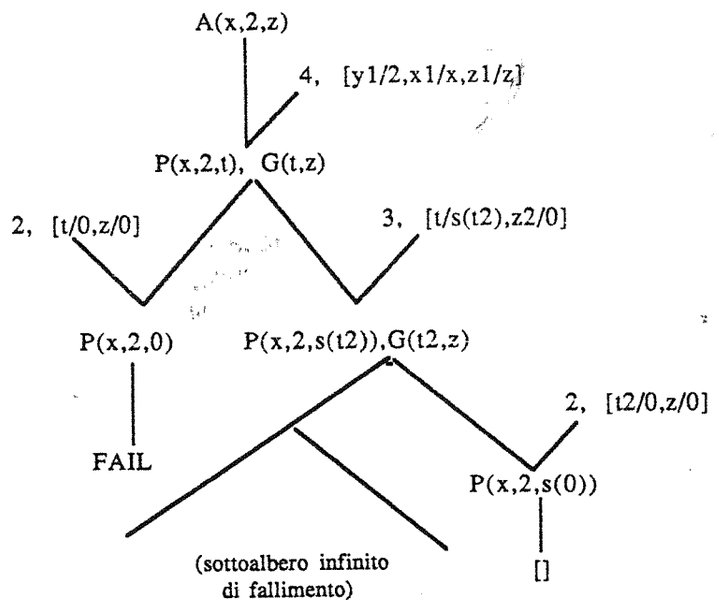
Vediamo un esempio: sia T il seguente programma:

1 P(x,y,1) :-
2 G(0,0) :-
3 G(s(t),z) :- G(t,z)
4 A(x,y,z) :- P(x,y,t), G(t,z)

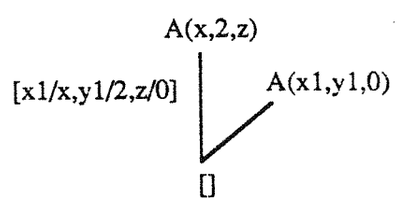
Il programma T' risultante dalla valutazione parziale di $T \cup \{A(x,y,z)\}$ è composto dalla seguente clausola:

A(x,y,0):-

Consideriamo la derivazione di $T \cup \{A(x,2,z)\}$ secondo la regola di computazione rightmost:



Usando come regola di computazione la rightmost il precedente albero è infinito. Adesso costruiamo la derivazione di T''∪{A(x,2,z)}:



La lunghezza della derivazione con successo di A(x,2,z) è minore in T' rispetto a T. Inoltre nell'albero SLD di T ∪ {A(x,2,z)} esistono infinite derivazioni di fallimento (anche adottando una regola fair il numero dei fallimenti infiniti può essere maggiore di 0), mentre in quella di T' ∪ {A(x,2,z)} esiste una sola derivazione che è di successo.

Notiamo infine che la derivazione in P' non è infinita, al contrario di quanto accade in P adottando la regola di computazione rightmost.

Bibliografia

[Aquilano-85] *Aquilano C., e Bocchetti P.* **Caratterizzazione di una teoria in clausole per una completa deduzione di informazioni negative.** Tesi di laurea, Dipartimento di Informatica, Università di Pisa, 1985.

[Ershov-77] *Ershov A.* **On the Partial Evaluation principle.** *Information Processing Letters* 6.2, 38-41, 1977.

[Fanti-86] *Fanti L., e Zanobetti S.* **Proprietà della Valutazione Parziale in Programmazione Logica.** Tesi di Laurea, Dipartimento di Informatica, Università di Pisa, 1986.

[Furukawa-86] *Furukawa K., e Takeuchi A.***Partial Evaluation of Prolog programs and its Application to Metaprogramming.**Information Processing 86, H.J. Kugler, Ed., Elsevier Science Publishers B.V., 1986.

[Komorowski-81] *Komorowski H.J.* **A specification of an abstract PROLOG machine and its application to partial evaluation.** Ph.D. Thesis 69, Linkoping University, Sweden.

[Ghelfo-87] *Ghelfo S., e Levi G.* **Un Valutatore Parziale per Metaprogrammi in un linguaggio logico con teorie multiple**, Atti GULP 1987, Torino.

[Levi-87] *Levi G., e Sardu G.* **Partial Evaluation of metaprograms in a "multiple worlds" logic language**, *in corso di pubblicazione su New Generation Computing*.

[Lloyd-84] *Lloyd J.W.* **Foundations of Logic Programming.** Springer Verlag, Symbolic Computation Series, 1984.

[Tamaky-84] *Tamaky H.,e Sato T.***Unfold/Fold Trasformation of Logic Programs.** Proc. Second International Logic Programming Conf., Upsaala 1984, 127-138.

[Venken-84] *Venken R.* **A Prolog meta-interpeter for Partial Evaluation and its application to source-to-source transformation and query**

optimization. Proc. ECAI-84: Advances in Artificial Intelligence (T. O'Shea, Ed.), North Holland, 1984, 91-100.

UN'ESTENSIONE DELL'INTERPRETE PROLOG CON MODULARITA' E CONCORRENZA

Paola Mello (*), *Antonio Natali* (*),
Aurelio Rossolini (*), *Francesco Russo* (**)

(*) DEIS - Facolta' di Ingegneria - Bologna
(**) ENIDATA - SOPEB - Bologna

Sommario

In questo lavoro si descrivono le estensioni apportate ad un interprete Prolog per consentire la definizione di programmi Prolog modulari e concorrenti in ambiente mono-processore. In particolare viene introdotto un insieme di primitive che consentono l' uso del Prolog come linguaggio di sistema. In questo modo la potenza espressiva del linguaggio e' sfruttata per la prototipazione di diversi "nuclei" di sistema e differenti ambienti di programmazione eliminando la necessita' di ricorrere ad altri linguaggi o strumenti. Le estensioni descritte sono state utilizzate quale supporto implementativo del modello CPU (Communicating Prolog Units), precedentemento proposto per la definizione di modularita' e concorrenza in Prolog.

Il lavoro e' stato parzialmente finanziato da ENIDATA S.p.A., CEE (Progetto Esprit P973) e MPI (40%).

INTRODUZIONE

Due concetti sembrano essere fondamentali per l'utilizzo del linguaggio Prolog quale linguaggio di sistema per la rapida prototipazione di differenti supporti a tempo di esecuzione e per la costruzione di ambienti di programmazione integrati: la modularita' e la concorrenza. Lo scopo principale del lavoro qui descritto e' stato quello di individuare ed implementare i meccanismi base che devono essere inseriti nel linguaggio Prolog per consentire tale possibilita', naturalmente mantenendo con Prolog piena compatibilita'.

L' obiettivo e' di fornire un insieme minimale di meccanismi tale da consentire di implementare in Prolog, in modo altamente flessibile, differenti politiche di combinazione fra moduli e di comunicazione e sincronizzazione fra processi.

Per quello che riguarda la modularita', soprattutto analizzando proposte che tendono ad estendere Prolog con concetti tipici della programmazione ad oggetti ([1], [2]), si e' arrivati alla conclusione che il meccanismo con cui costruire