gruppo ricercatori e utenti
di
logic programming

atti del primo convegno nazionale sulla programmazione logica

genova
12-14 marzo 1986

realizzato con il contributo della
BANCA POPOLARE DI BERGAMO

G.U.L.P.
Gruppo ricercatori ed utenti di
logic programming

Atti del
Primo Convegno Nazionale
sulla
PROGRAMMAZIONE LOGICA

Genova 12-14 Marzo 1986

a cura di
Giuliana Dettori
Istituto per la Matematica Applicata
Consiglio Nazionale delle Ricerche
Genova

Realizzato con il contributo della BANCA POPOLARE DI BERGAMO

G.U.L.P
Gruppo ricercatori ed utenti di
logic programming

Primo Convegno Nazionale sulla Programmazione Logica

Genova, 12-14 marzo 1986

Presidente del Convegno

Giuliana Dettori (I.M.A., C.N.R., Genova)

Comitato di Programma

Giorgio Levi, presidente (Dip. Informatica, Univ. di Pisa)
Giovanni Adorni (D.I.S.T., Univ. di Genova)
Giuliana Dettori (I.M.A., C.N.R., Genova)
Pietro Jalamoff (Scuola Sup. Reiss Romoli, L'Aquila)
Luigi Marcolungo (Ist. Studi Internazionali, Univ. di Padova)
Maurizio Martelli (C.N.U.C.E., C.N.R., Pisa)
Leonardo Roncarolo (ELSAG, Genova)
Gianfranco Rossi (Dip. Informatica, Univ. di Torino)
Umberto Rugani (INTECS, Pisa)
Genny Tortora (Dip. Informatica e Applicazioni, Univ. di Salerno)

Hanno partecipato alla organizzazione del Convegno

Ist. per la Matematica Applicata, Consiglio Nazionale delle Ricerche, Genova
Dipartimento di Informatica, Università di Pisa
Dip. di Informatica, Sistemistica e Telematica, Università di Genova
Elettronica S.Giorgio, ELSAG s.p.a., Genova
Istituto di Studi Internazionali, Università di Padova

Sede del Convegno

Banca Popolare di Bergamo, Via Fieschi 11, Genova

PREMESSA

Benvenuti al Primo Convegno sulla Programmazione Logica.
Questo convegno rappresenta il primo incontro scientifico organizzato dal G.U.L.P. - "Gruppo ricercatori ed utenti di logic programming", costituito di recente con lo scopo di raccogliere quanti in Italia utilizzano strumenti e metodologie della programmazione logica o svolgono attività di ricerca in questo campo.
La quantità di adesioni pervenute al gruppo (circa un centinaio) e il numero di lavori (35) inviati per il convegno provenienti sia da ambienti accademici (19), sia dall'industria (10), sia frutto di collaborazione tra ambienti diversi (6), sottolineano il crescente interesse per la programmazione logica.

La selezione delle proposte da parte del comitato di programma è stata fatta in modo da conferire al convegno il carattere di giornate di lavoro dove sia possibile mettere a confronto le esperienze maturate in ambienti diversi, sia sulla ricerca che sulle applicazioni della programmazione logica.
I 30 articoli accettati sono stati suddivisi in sette aree:
linguaggi e architetture, sistemi esperti, ambienti di programmazione, rappresentazione della conoscenza, basi di dati, linguaggio naturale, didattica.

Il convegno è stato pensato e organizzato prima che il G.U.L.P. potesse essere costituito legalmente, quindi con un comitato direttivo provvisorio e senza soci. Desidero perciò scusarmi per le eventuali carenze che potrete riscontrare nell'organizzazione dei lavori o in questi atti.

Desidero infine ringraziare quanti hanno collaborato per la buona riuscita di questa attività, in particolare:

I membri del comitato di programma, con il presidente Giorgio Levi, per il lavoro svolto per la selezione dei lavori; Giovanni Adorni, Luigi Marcolungo e Leo Roncarolo per l'aiuto nell'organizzazione.

La Banca Popolare di Bergamo che ha ospitato il convegno nei suoi locali, ha offerto la stampa di questi atti ed un rinfresco di benvenuto ai partecipanti.

L'Elettronica S. Giorgio - ELSAG, l'Istituto per la Matematica Applicata del C.N.R. e la SOI Informatica, che hanno contribuito a finanziare il convegno.

Patrizia Asirelli dell'I.E.I. di Pisa che ha seguito con pazienza l'iter burocratico per la registrazione del G.U.L.P.

Roberto Barbuti, del Dipartimento di Informatica di Pisa, che ha preparato la copertina di questi atti.

Gli autori dei lavori e tutti i partecipanti al convegno che con la loro partecipazione hanno permesso la realizzazione di questo incontro.

Giuliana Dettori
Presidente del Convegno

INDICE

LA TEORIA INTUIZIONISTICA DEI TIPI: UNA INTRODUZIONE ED IL TEOREMA DI NORMALIZZAZIONE.

A. Bossi - S. Valentini

Istituto di Algebra e Geometria
via Belzoni, 7. Padova

ABSTRACT

Scopo del nostro attuale lavoro di ricerca e' fornire prove dettagliate delle principali proprieta' logiche e computazionali della teoria intuizionistica dei tipi di P. Martin-Lof, come ad esempio la terminazione di ogni programma corretto sviluppato al suo interno. Questa ed altre proprieta' conseguono dalla normalizzabilita' delle derivazioni nella teoria dei tipi. La prova di normalizzazione e' stata da noi ottenuta con un metodo alla Tait (vedi (1)).

In questa nota presentiamo una breve descrizione informale della teoria ed illustriamo le idee che portano al teorema di normalizzazione.

INTRODUZIONE

E' ormai ampiamente riconosciuta l'importanza di basi teoriche a supporto di una attivita' di programmazione sempre meno legata alle caratteristiche di una specifica macchina e sempre piu' interessata all'uso di strumenti teorici per la sintesi e l'analisi dei programmi. Tra le teorie logico matematiche utilizzabili a questo scopo la Matematica Costruttiva ed in particolare la Teoria Intuizionistica dei Tipi (2) promette risultati concreti. Noi abbiamo esaminato questo strumento logico seguendo due linee principali. La prima e' stata relativa alla sintassi (Teoria delle Espressioni con Arieta' (3)) per la scrittura delle formule ben formate della teoria dei tipi; la seconda, in fase di completamento, e' lo studio delle sue principali proprieta' formali. Alcune di queste vengono spesso riportate come cosa nota o evidente ma, nella nostra esperienza, una dimostrazione rigorosa e' tutt'altro che banale. Inoltre una dimostrazione non raggiunge solo il suo scopo primario evidente ma e' anche un mezzo per comprendere meglio il sistema stesso.

Nel seguito daremo una breve presentazione informale della teoria ed

enunceremo il teorema di normalizzazione che abbiamo sviluppato per dimostrare esplicitamente le sue piu' importanti proprieta' logiche e computazionali.

LA TEORIA DEI TIPI

La teoria intuizionistica dei tipi e' stata originariamente sviluppata da P. Martin-Lof come notazione per una precisa codifica della matematica costruttiva. In (4) lo stesso autore suggerisce che la teoria dei tipi puo' essere altrettanto bene considerata un linguaggio di programmazione: "If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods that is the computer's duty to execute, then it no longer seems possible to distinguish the discipline of programming from constructive mathematics". Lo stile di programmazione che si impone naturalmente con l'uso di questo linguaggio si puo' schematizzare nei due passi: 1) specificare il problema al livello desiderato di astrazione; 2) sviluppare, nello stesso linguaggio, la prova (costruttiva) che il problema e' risolvibile. Alla fine si avra' il programma che soddisfa la specifica data.

La correttezza di un programma scritto in teoria dei tipi e' pertanto formalmente dimostrata contemporaneamente al suo sviluppo. Non vi e' una derivazione automatica del programma ma il processo di "problem-solving" viene facilitato dalla ricca tipizzazione del linguaggio che permette di rappresentare ogni ragionevole specifica con un tipo, di usare una progettazione modulare e di procedere per raffinamenti successivi.

Un tipo si puo' ottenere, a partire dai tipi base, cioe' i Tipi Finiti (Tipi Enumerazione) o N (il tipo dei numeri naturali), tramite l'uso di costruttori quali il Prodotto Cartesiano, l'Unione Disgiunta ed i Buoni Ordini. Particolari regole di inferenza, formulate nello stile della Deduzione Naturale alla Gentzen per il calcolo predicativo, permettono di ottenere giudizi quali "A e' un tipo" (notazione A type) oppure "A e B sono tipi uguali" (A=B), "a e' un oggetto del tipo A" (a ∈ A), "a e b sono oggetti uguali del tipo A" (a=b ∈ A). Queste sono le quattro forme di giudizio che, in accordo con la corrispondenza tra proposizioni e tipi (Curry 1958 e Howard 1969) e l'interpretazione delle proposizioni come problemi o scopi (Kolmogorov 1932), hanno differenti letture se sostituiamo la parola tipo con quella di proposizione o di problema e la parola oggetto con la parola dimostrazione o programma. Percio', per esempio, la terza forma di giudizio (a e' un elemento del tipo A) puo' essere letta anche come: "a e' una prova della proposizione A" oppure "a e' un programma che risolve il problema A".

Analizziamo piu' in dettaglio le possibili forme di derivazione del giudizio a ∈ A. Innanzitutto esso puo' essere ottenuto usando una regola di introduzione che prescrive la forma canonica degli elementi di A. Consideriamo, ad esempio, il problema A-->B (cioe' quello di risolvere il problema B sotto l'assunzione che il problema A sia solubile) e supponiamo di avere un metodo b(x) per risolvere B a patto che x risolva A (cioe' sotto l'assunzione Cx = x ∈ A). Come e' ovvio aspettarsi potremo allora introdurre la funzione $\lambda(b)$, utilizzando la regola di inferenza:

$$\frac{b(x) \in B \ (\ x \in A\ )}{\lambda(b) \in A\text{-->}B} \qquad (\text{ --> introduzione })$$

$\lambda(b)$ e' una soluzione canonica del problema A-->B ovvero, ritornando alla prima lettura, un elemento canonico del tipo A-->B.

Notiamo che $\lambda(b)$ e' un elemento canonico anche quando b(x) non lo e'. Infatti e' la costante piu' esterna che individua sintatticamenete gli elementi canonici. Vale anche la pena di osservare che la sintassi sottostante la teoria dei tipi e' data dalle espressioni con arieta' e che in questo ambiente la costante lambda e' usata per denotare funzioni come oggetti del tipo A-->B, mentre l'astrazione di x dall'espressione e viene denotata con (x)e.

Oltre alle regole di introduzione, il giudizio a ∈ A, puo' essere derivato usando regole di eliminazione. In questo caso la soluzione si presenta come un metodo che, quando eseguito, fornisce un valore, che e' un elemento canonico, del tipo considerato. Ad esempio, l'induzione e' la regola di eliminazione associata con il tipo N dei numeri naturali. Sia C(z) (z ∈ N) un problema con dominio sui numeri naturali (poiche' non abbiamo alcun bisogno di originalita' su questo punto possiamo considerare il problema di trovare il fattoriale di z). Supponiamo ora di avere il numero naturale c (cioe' c ∈ N), di avere un elemento d di C(0) (cioe' d=1, la soluzione del problema di trovare 0!), e di saper trovare un elemento e(x,y) di C(succ(x)) sapendo che x e' un numero naturale e che y e' un elemento di C(x) (cioe' e(x,y)=y*succ(x) se y e' la soluzione del problema di trovare x!). In queste ipotesi e' chiaro che noi abbiamo un metodo (la recursione) che ci permette di risolvere il problema C(c) (nel nostro esempio di trovare c!). Denoteremo questo metodo con Rec(c,d,e) ed esso sara' un elemento non-canonico del tipo C(c).

$$\frac{c \in N \qquad d \in C(0) \qquad e(x,y) \in C(succ(x)) \ (x \in N, y \in C(x))}{Rec(c,d,e) \in C(c)}$$

La sua esecuzione fornira' un valore (canonico) di C(c). L'esecuzione e' definita tramite regole di computazione che forniscono la semantica operazionale per i programmi scritti in teoria dei tipi.

Un elemento canonico ha se stesso come valore mentre le regole di computazione per gli elementi non-canonici seguono il significato suggerito dalla regola di eliminazione che produce l'elemento non-canonico. Ad esempio il metodo denotato da Rec(c,d,e) e' il seguente: calcolare il valore di c (e poiche' c ∈ N si otterra' 0 o succ(c'), il successore di un qualche numero naturale c'); se si e' ottenuto 0 allora il risultato e' il valore di d altrimenti il risultato e' il valore di e(c',Rec(c',d,e)). Tutto cio' e' espresso dalle regole di computazione:

$$0 ==> 0 \qquad\qquad succ(a) ==> succ(a)$$

$$\frac{c ==> 0 \qquad d ==> g}{Rec(c,d,e) ==> g} \qquad\qquad \frac{c ==> succ(c') \qquad e(c',Rec(c',d,e)) ==> g}{Rec(c,d,e) ==> g}$$

Una regola di computazione descrive solo un passo del processo di valutazione di un programma. L'intera valutazione puo' essere rappresentata da un albero che chiamiamo albero di valutazione. Per dare un esempio significativo introduciamo anche la regola di eliminazione per il tipo A --> B e la corrispondente computazione:

$$\frac{c \in A \text{ --> } B \qquad a \in A}{Ap(c,a) \in B} \qquad\qquad \frac{c ==> \lambda(b) \qquad b(a) ==> e}{Ap(c,a) ==> e}$$

La valutazione completa di

Ap(Rec(succ(0), λ((x)x),(u,v) λ((x)succ(Ap(v,x)))), 2) e':

succ(0)==>succ(0) λ((x)succ(Ap(Rec(0, λ((x)x),..),x)))
==> λ((x)succ(Ap(Rec(0, λ((x)x),..),x)))

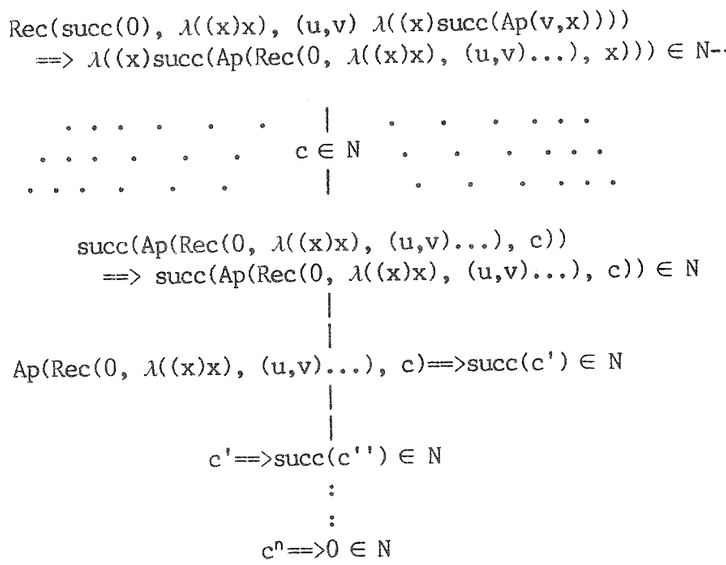
Rec(succ(0), λ((x)x),(u,v) λ((x)succ(Ap(v,x))))
==> λ((x)succ(Ap(Rec(0, λ((x)x),..),x)))
. .
succ(Ap(Rec(0, λ((x)x),..),2))
==>succ(Ap(Rec(0, λ((x)x),..),2))

Ap(Rec(succ(0), λ((x)x),(u,v) λ((x)succ(Ap(v,x)))),2)
==>succ(Ap(Rec(0, λ((x)x),(u,v) λ((x)succ(Ap(v,x)))),2))

VERSO IL TEOREMA DI NORMALIZZAZIONE

E' chiaro che un programma e' valutabile, cioe' ha un valore, quando il suo albero di valutazione e' ben fondato. Poiche' questo albero racchiude solamente la conoscenza della sintassi del programma non possiamo aspettarci che questa sia sufficiente per provare che ogni programma derivabile in teoria dei tipi ha un valore. In effetti basta pensare che l'albero di valutazione per il ben noto programma di Church:
Ap(λ((x) Ap(x,x)), λ((x) Ap(x,x))), che e' sintatticamente corretto, non e' ben fondato.

Per sviluppare una dimostrazione che l'esecuzione di ogni programma costruito in teoria dei tipi termina dobbiamo prendere in considerazione tutta l'informazione contenuta nella derivazione del programma stesso. Le regole di computazione rispecchiano le premesse delle regole di eliminazione sugli elementi non-canonici ma si fermano su quelli canonici ovvero quando viene usata una regola di introduzione. D'altra parte, a volte non ha senso chiedere di valutare le componenti (cioe' gli elementi che compaiono nelle premesse) di un elemento canonico, come ad esempio l'espressione b(x) nel caso di λ(b). Tuttavia dalle premesse della regola di introduzione per λ(b) sappiamo che b(x) ∈ B se x ∈ A e pertanto la domanda corretta e' se b(a) con a ∈ A e' valutabile. Per cogliere queste idee noi introduciamo la nozione di albero di computazione, cioe' iteriamo il processo di valutazione anche sulle istanze delle premesse di una regola di introduzione. Ad esempio l'albero di computazione di Rec(succ(0), λ((x)x),(u,v) λ((x)succ(Ap(v,x)))) e'



La definizione formale di cosa significa: "il giudizio a ∈ A e' computabile" formalizza il concetto "l'albero di computazione per a ∈ A e' ben fondato". Non avrebbe senso dare qui una definizione rigorosa in quanto essa richiede una conoscenza del formalismo della teoria dei tipi che va ben al di la' della presentazione che abbiamo dato della teoria in queste pagine; tuttavia possiamo dare una migliore intuizione della definizione di computabile come segue: (indicheremo tra " " i termini che andrebbero ulteriormente definiti)

1) se nell'espressione a non compaiono variabili allora il giudizio a ∈ A, derivato dalle assunzioni Cx_1 ,..,Cx_n , si dice computabile se i) a si valuta in un elemento canonico g ii) appartenente ad un tipo "uguale" ad A iii) e tale che le "componenti" di g siano computabili.

2) se nell'espressione a compaiono solo le variabili y_1 ,..,y_m allora il giudizio a ∈ A, derivato dalle assunzioni Cy_1 ,..,Cy_m, Cx_1,..,Cx_n , si dice computabile se ogni giudizio ottenuto da a ∈ A sostituendo "correttamente" le y_i con espressioni computabili e' computabile.

Osserviamo che dalla definizione segue immediatamente che se a ∈ A e' computabile ed a non ha variabili allora a ha un valore.

Il principale teorema sulla computabilita' e' il seguente:

Teorema : Se a ∈ A e' un giudizio derivabile nella teoria dei tipi dalle assunzioni C_1,..,C_n allora a ∈ A e' computabile.

La dimostrazione si ottiene per induzione sulla lunghezza della

derivazione di a ∈ A dalle assunzioni C_1,..,C_n. Essa, chiaramente, procede per casi a seconda dell'ultima regola usata nella derivazione. A titolo di esempio, vediamo come ci si puo' convincere che il giudizio λ(b) ∈ B derivabile dalle assunzioni C_1,..,C_n, e' computabile, sapendo, per ipotesi induttiva, che lo e' il giudizio b(x) ∈ B derivabile dalle assunzioni C_1 ,..,C_n , x ∈ A. Ora λ(b) ==> λ(b), che e' un elemento canonico di un tipo, B stesso, che evidentemente risulta uguale a B e le sue componenti sono computabili per ipotesi induttiva. La prova per tutte le altre regole di introduzione si ottiene in modo analogo. Per trattare le regole di eliminazione e' essenziale la definizione di sostituzione corretta e l'uso delle regole di uguaglianza che non abbiamo introdotto in questa presentazione della teoria. La traccia della dimostrazione per la --> eliminazione, quando in Ap(c,a) non compaiono variabili, e' la seguente. Per ipotesi induttiva: c∈A --> B ed a∈A sono computabili; dalla prima si puo' ricavare che b(x) ∈ B e' computabile e quindi che b(a) ∈ B e' computabile, essendo x:=a una sostituzione corretta. Da questo, poiche' il valore di Ap(c,a) e' per definizione il valore di b(a), si puo' ricavare che Ap(c,a)∈A --> B e' computabile.

Seguono immediatamente alcuni corollari. Se diciamo che il programma a e' corretto rispetto alla specifica A quando il giudizio a ∈ A e' dimostrabile (= derivabile senza assunzioni) allora si ha che ogni programma corretto termina. Inoltre segue anche che il valore di a e' un elemento canonico dello stesso tipo cui appartiene a. Questo fatto assicura che le regole di computazione conservano la correttezza.

Vale infine la pena di osservare che da un punto di vista puramente logico il precedente teorema e' analogo ad un teorema di normalizzazione e pertanto assicura la consistenza della teoria dei tipi: non puo' essere derivata, al suo interno, alcuna prova dell'assurdo.

BIBLIOGRAFIA

(1) Troelstra, A.S. Metamathematical Investigations of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics 344, Springer-Verlag 1973.

(2) Martin-Lof, P. Intuitionistic Type Theory, Bibliopolis, Napoli, 1984.

(3) Bossi, A. e Valentini, S. The expressions with arity. Rapporto interno. Giugno 1985.

(4) Martin-Lof, P. Constructive Mathematics and computer programming. Mathematical logic and programming languages. C. Hoare and J. Shepherdson (Eds.), Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985, pp. 167-184.

UN METODO DI RISOLUZIONE PER IL CALCOLO DEI PREDICATI MODALE

Marta Cialdea

Université Paul Sabatier - Toulouse (France) (*)
Laboratorio di Intelligenza Artificiale - Selenia S.p.A. - Roma

ABSTRACT

In questo lavoro viene presentato un metodo di risoluzione per un sistema di logica modale, che risulta dall'estensione all'intera logica dei predicati dei metodi conosciuti per la risoluzione modale proposizionale. Tale risultato rappresenta un contributo all'estensione della programmazione logica al linguaggio modale.

INTRODUZIONE

Negli ultimi anni le logiche non classiche si sono dimostrate uno strumento adeguato per formalizzare diversi problemi nell'ambito dell'informatica e dell'intelligenza artificiale. Le logiche intensionali, in particolare, presentano uno spettro di applicazioni notevolmente ampio, permettendo di esprimere concetti non estensionali quali la necessità (logiche modali), l'obbligo (logiche deontiche), i riferimenti temporali (logiche temporali), le conoscenze di diversi soggetti (logiche epistemiche).

La logica modale ([13]), la più antica delle logiche non classiche, è, in un certo senso, il quadro di riferimento o il nucleo comune di tutte le logiche intensionali. Essa infatti estende il linguaggio della logica classica per mezzo dell'operatore proposizionale di necessità e del suo duale, la possibilità. Stabilendo quali debbano essere le proprietà di cui godono tali concetti, ed elimindando così le ambiguità che essi possiedono nel linguaggio comune, si possono ottenere diversi sistemi modali, ciascuno dei quali intende formalizzare una diversa accezione dei concetti modali. Gli operatori di base delle altre logiche intensionali possono generalmente essere considerati delle estensioni o delle precisazioni dei due concetti modali fondamentali, nel senso che essi godono delle stesse proprietà che possono essere attribuite alla necessità o alla possibilità.

Nel dominio dell'informatica teorica, i concetti intensionali si sono rivelati adatti a trattare problemi di semantica dei linguaggi e per la dimostrazione di proprietà dei programmi. Con questo scopo sono state sviluppate delle estensioni molto ricche della logica modale, come la logica algoritmica e la logica dinamica, e sono state utilizzate diverse versioni della logica temporale per specificare e dimostrare proprietà di programmi paralleli.

(*) I risultati esposti in questo articolo sono estratti dalla Tesi di Dottorato svolta presso l'Univeristé Paul Sabatier di Toulouse (Francia) negli anni 1983-1985.

Nell'ambito dell'Intelligenza Artificiale, le logiche intensionali trovano applicazione in numerosi settori, quali la comprensione del linguaggio naturale, dove la logica epistemica e la logica temporale permettono di rappresentare la struttura profonda di alcune espressioni del linguaggio naturale; oppure nella formalizzazione di tipi di ragionamento non classici, quali il ragionamento non monotono e il ragionamento temporale.

La ricchezza espressiva delle logiche intensionali puo' comunque essere sfruttata appieno soltanto se si dispone per esse di metodi efficienti di dimostrazione automatica. Un obiettivo naturale consiste dunque nei tentativi di estendere alle logiche intensionali i metodi di dimostrazione automatica più efficienti conosciuti per la logica classica, ed in particolare del metodo di risoluzione ([15]), sul quale si basa il linguaggio Prolog.

Per la parte proposizionale di diversi sistemi di logica intensionale sono stati definiti dei procedimenti di risoluzione ([6],[8],[3],[1]) e ne sono state realizzate alcune implementazioni ([9],[16],[14],[5]). Attualmente, presso l'università di Tolosa, si stanno studiando le modalità per realizzare un'estensione modale del Prolog, il linguaggio Molog ([7]), la cui prima versione è stata implementata in Prolog su un DPS-8 ([11]).

I metodi di risoluzione già noti per le logiche modali sono tuttavia limitati alla parte proposizionale dei sistemi considerati, oppure alle formule in forma prenessa. In questo lavoro proponiamo una soluzione al "problema dei quantificatori" in logica modale ed una estensione alla logica dei predicati dei metodi di risoluzione proposizionale esistenti.

Il nostro punto di partenza è costituito da due risultati fondamentali: il primo consiste naturalmente nei sistemi di risoluzione modale proposizionale, l'altro è il teorema di Herbrand per la logica classica ([12]). E' questo infatti il risultato che, nella logica classica, permette di "elevare" la risoluzione proposizionale alla risoluzione generale, ed è questo dunque il risultato di cui occorre trovare un'estensione modale perché sia possibile effettuare un salto analogo in logica modale.

In questo lavoro ci limiteremo a considereare un sistema modale molto semplice, il sistema Q, che è una sotto-teoria dei sistemi modali più noti T, S4 e S5 di Lewis ([13]).

IL SISTEMA MODALE Q

Poiché non vogliamo presupporre nel lettore una particolare familiarità con la logica modale, in questa sezione definiremo in dettaglio la sintassi e la semantica del sistema Q.

Il linguaggio logico del sistema Q consiste di tutti i simboli logici della logica classica e dei due operatori modali $\Box$ (necessità) e $\Diamond$ (possibilità). Le formule modali includono tutte le formule classiche e le formule della forma $\Box A$ e $\Diamond A$, dove A è una formula (modale).

La teoria modale Q è costituita da tutti gli assiomi e le regole di inferenza della logica classica e:

(Q1) $\quad \Box(A \to B) \to (\Box A \to \Box B)$

(Q2) $\quad \Box A \to \Diamond A$

Regola di necessitazione: $\dfrac{A}{\Box A}$

La semantica del sistema Q è definita in termini di modelli di Kripke come segue. Una **struttura modale** per un linguaggio modale L è una quadrupla $I = (W,R,D,v)$, dove:

W è un insieme non vuoto (l'insieme dei "mondi possibili");
R è una relazione binaria definita su W, la relazione di "accessibilità" tra i mondi possibili, tale che per ogni $w \in W$ esiste $w' \in W$ tale che $R(w,w')$;

D è una funzione che assegna a ogni $w_i \in W$ un dominio di oggetti $D(w_i)$ in modo tale che, se $R(w_i,w_j)$, allora $D(w_i) \subseteq D(w_j)$; nel seguito, D_∞ denoterà l'unione degli insiemi $D(w)$:

$$D_\infty = \bigcup_{w \in W} D(w):$$

v è una funzione che definisce una interpretazione classica per ogni $w \in W$: essa è definita sugli elementi del linguaggio, in modo tale che:
- se c è una costante individuale, allora $v(c) \in D_\infty$;
- se f è un simbolo funzionale, allora $v(f)$ è una funzione definita su D_∞, tale che per ogni $d_1,\dots,d_n \in D_\infty$, $v(f)(d_1,\dots,d_n) \in D_\infty$;
- se p è un simbolo predicativo n-ario, allora $v(p) \subseteq \{(w_i,d_1,\dots,d_n) : w_i \in W,\ d_1,\dots,d_n \in D_\infty\}$.

Una **interpretazione** di un linguaggio modale L è una coppia (I,w_o), dove I è una struttura modale per L e w_o è un elemento di W (il "mondo attuale").

La relazione di **soddisfacibilità** tra le coppie (I,w), dove I è una struttura modale e $w \in W$, e le formule di L, denotata da $(I,w) \models A$, è definita aggiungendo le due clausole seguenti alla definizione induttiva classica:

$(I,w) \models \Box A$ sse per ogni w' tale che $R(w,w')$, $(I,w') \models A$;
$(I,w) \models \Diamond A$ sse esiste w' tale che $R(w,w')$ e $(I,w') \models A$.

Una formula A è vera in una interpretazione $M = (I,w_o)$ se e solo se $(I,w_o) \models A$; in tal caso diciamo anche che M è un **modello** di A. Una formula A è **soddisfacibile** se esiste una interpretazione M in cui A è vera.

Il sistema Q è uno dei sistemi modali più poveri e non ha, direttamente, una grande importanza applicativa.(1) La scelta di tale sistema è dovuta, da una parte, a una motivazione "euristica", che porta ad affrontare in prima istanza le situazioni più semplici, con lo scopo di sperimentare metodologie di soluzione che possano essere poi adattate alle situazioni più complesse.

D'altro canto, il sistema contiene già per intero il problema dei quantificatori in logica modale, in quanto i sistemi T, S4 ed S5 differiscono da Q soltanto nella parte proposizionale.

Osserviamo che la formula di Barcan, $\forall x \Box A(x) \rightarrow \Box \forall x A(x)$, che, dal punto di vista semantico, implica l'identità di tutti i domini di oggetti associati ai mondi possibili, non è un teorema di Q. Nemmeno la formula $\forall x \Diamond A(x) \rightarrow \Diamond \forall x A(x)$ è un teorema di Q. Al contrario, sono teoremi le formule $\Box \forall x A(x) \rightarrow \forall x \Box A(x)$ e $\Diamond \forall x A(x) \rightarrow \forall x \Diamond A(x)$, che determinano, nella semantica, il fatto che i domini associati ai mondi possibili possono soltanto crescere: se $R(w,w')$, allora $D(w) \subseteq D(w')$.

Simmetricamente, sono teoremi di Q le leggi per il quantificatore esistenziale $\exists x \Box A(x) \rightarrow \Box \exists x A(x)$ e $\exists x \Diamond A(x) \rightarrow \Diamond \exists x A(x)$, mentre non sono teoremi le formule $\Box \exists x A(x) \rightarrow \exists x \Box A(x)$ e $\Diamond \exists x A(x) \rightarrow \exists x \Diamond A(x)$.

Come si puo' dunque osservare, un quantificatore in una formula modale ha un significato che dipende anche dal suo **grado modale**, cioè dal numero (e in alcuni sistemi modali dalla qualità) degli operatori modali che lo governano. Per esempio le due formule $\Diamond \exists x p(x)$ e $\forall x \Box \neg p(x)$ non sono in contraddizione l'una con l'altra, mentre lo sono $\exists x \Diamond p(x)$ e $\forall x \Box \neg p(x)$.

La semplicità del sistema Q per quel che riguarda il problema dei quantificatori risiede nel fatto che in tale sistema ogni operatore modale ha, in un certo senso, lo stesso "peso" per i quantificatori che occorrono nel suo scopo, senza che sia necessario fare alcuna distinzione tra $\Box$ e $\Diamond$, né considerare la qualità di operatori modali annidati. Cio' significa che possiamo identificare per il sistema Q una semplice legge che regola la possibilità di spostare un quantificatore all'interno o all'esterno di un operatore modale: "il quantificatore esistenziale puo' spostarsi all'interno degli operatori modali, ma non all'esterno; il quantificatore universale puo' spostarsi all'esterno degli operatori modali, ma non all'interno".

(1) Il sistema Q sembra essere preso in considerazione soltanto in alcuni studi di logica deontica, dove l'operatore di necessità è interpretato come "è obbligatorio" ([10],[2]).

Nel seguito, adotteremo in generale le notazioni logiche standard. Inoltre useremo le seguenti convenzioni. Se A è una formula e B è una sottoformula di A, cio' sarà a volte evidenziato indicando A con A[B]. Inoltre, se A' è la formula ottenuta da A sostituendo la sottoformula B con C, allora A' sarà indicata da A[C].

IL SISTEMA DI FORMULE CON LIVELLI E L'UNIFICAZIONE MODALE

Il metodo di risoluzione generale per il sistema Q è dato dalla fusione del metodo proposizionale con un concetto di unificazione modale, ottenuto imponendo alcune restrizioni alle operazioni di sostituzione.

Come si è visto, in logica modale non è realizzabile la trasformazione di qualsiasi formula in una equivalente che sia in forma prenessa, dato che gli operatori modali bloccano la possibilità di spostare i quantificatori all'esterno. I quantificatori non possono dunque essere eliminati in maniera semplice come nella logica classica, per preparare un insieme di formule alla risoluzione: in una formula modale è importante ricordare il grado modale dei suoi quantificatori.

Definiremo allora una opportuna traduzione T(A) di una formula modale A, nella quale i quantificatori vengono eliminati, ma l'informazione relativa al loro grado modale viene conservata grazie all'attribuzione di **livelli** a variabili, costanti e simboli funzionali. La traduzione di una formula è dunque una formula senza quantificatori, in cui ogni variabile, costante e simbolo funzionale puo' essere etichettato da un numero naturale (che sarà indicato ad esponente del simbolo stesso), uguale al grado modale dei quantificatori cui corrisponde.

Per la definizione della traduzione T e del sistema di risoluzione modale avremo bisogno delle seguenti definizioni.

Una formula A è in **forma normale** sse:
- A non contiene alcun segno di implicazione,
- nessun simbolo logico in A occorre nello scopo di una negazione, e
- le variabili vincolate che occorrono in A sono nominate in modo differente l'una dall'altra.

Si puo' mostrare facilmente che ogni formula modale A puo' essere trasformata in una formula equivalente A' in forma normale. Nel seguito, considereremo sempre formule in forma normale: per "formula" intenderemo d'ora innanzi "formula in forma normale".

Le formule atomiche e le negazioni di formule atomiche saranno chiamate **letterali**.

La definizione della traduzione T richiede inoltre la definizione della funzione seguente, definita sull'insieme delle espressioni che possono contenere dei simboli etichettati da un livello (le N-espressioni):

γ è una funzione da N-espressioni a N-espressioni, tale che $\gamma(E)$ è l'espressione ottenuta da E sostituendo ogni simbolo s^n con s^{n+1}, cioè aumentando di 1 il livello dei simboli cui è stato assegnato un livello.

Nel seguito, se E è una espressione, $E\{t'/t\}$ denoterà l'espressione ottenuta da E sostituendo ogni occorrenza del simbolo t con t'.

La traduzione T è allora definita per induzione come segue:

1) $T(P) = P$ se P è un letterale
2) $T(A \wedge B) = T(A) \wedge T(B)$
3) $T(A \vee B) = T(A) \vee T(B)$
4) $T(\forall x A(x)) = T(A(x))\{x^0/x\}$
5) $T(\exists x A(x)) = T(A(c))\{f_x^0(y_1,\ldots,y_n)/c\}$
   dove c è una costante nuova e $f_x^0(y_1,\ldots,y_n)$ è il termine funzionale per x - dove $y_1,\ldots,y_n$ sono tutte le variabili libere in $\exists x A(x)$ - , con la funzione f_x etichettata dal livello 0.
6) $T(\Box A) = \Box(\gamma(T(A)))$
7) $T(\Diamond A) = \Diamond(\gamma(T(A)))$

Per esempio, $T(\forall x \Diamond \exists y\ p(x,y)) = T(\Diamond \exists y\ p(x,y))\{x^0/x\}$
$= \Diamond(\gamma(T(\exists y\ p(x,y))))\{x^0/x\}$
$= \Diamond(\gamma(T(p(x,c))\{f_y^0(x)/c\}))\{x^0/x\}$
$= \Diamond(\gamma(p(x,c)\{f_y^0(x)/c\}))\{x^0/x\}$
$= \Diamond(\gamma(p(x,f_y^0(x))))\{x^0/x\}$
$= \Diamond(p(x,f_y^1(x)))\{x^0/x\}$
$= \Diamond(p(x^0,f_y^1(x^0)))$

Nel seguito, i simboli senza livello saranno considerati di livello 0.

In [4] viene definita una opportuna nozione di soddisfacibilità per le formule con livelli, la N-soddisfacibilità, e si dimostra che una formula modale A è soddisfacibile se e solo se T(A) è N-soddisfacibile.

La nozione di sostituzione e le regole di risoluzione saranno in seguito formulate in termini di formule con livelli (o N-formule), in modo tale che un insieme S di formule modali è insoddisfacibile se e solo se T(S) è refutabile. Dunque, non si dovrà più tener conto dei quantificatori, ma solo dei livelli delle variabili e dei simboli costanti e funzionali.

Ricordando la semplice regola che determina quando è possibile scambiare di posto i quantificatori con gli operatori modali, la seguente nozione di sostituzione per espressioni con livelli risulta naturale (le sostituzioni saranno denotate da lettere greche minuscole: θ, σ, μ, ...).

Una **sostituzione** $\{t_1/x_1,\ldots,t_n/x_n\}$ è definita come nella logica classica, eccetto per il fatto che ogni variabile x_i e ogni simbolo in t_i puo' essere etichettato da un livello e, per ogni i, se n è il livello di x_i, allora per ogni simbolo s^m che occorre in t_i, m è minore o uguale a n. In altre parole, non è ammesso sostituire una variabile di livello n con un termine che contenga dei simboli di livello maggiore di n. Se E è un'espressione, la notazione $E\theta$ indicherà il risultato dell'applicazione della sostituzione θ all'espressione E.

In [4] viene dimostrato il seguente risultato, fondamentale per giustificare tale nozione di sostituzione:

Teorema. Se A è una formula, θ una sostituzione e A' è tale che $T(A') = T(A)\theta$, allora A' è vera in ogni modello di A.

Le nozioni di **unificatore** di un insieme di espressioni, insieme **unificabile** e **unificatore più generale (upg)** sono definite come nella logica classica. Nel seguito, $\emptyset$ denoterà la formula vuota, o il FALSO.

IL METODO DI RISOLUZIONE PER IL SISTEMA Q

Il risolvente di due N-formule C_1 e C_2 è definito utilizzando l'operazione ausiliaria $Z_\theta[C_1;C_2]$, dove θ è una sostituzione, definita induttivamente come segue.

(i) $Z_\theta[P_1;P_2] = \emptyset$ se θ è un upg di $\{P_1,\neg P_2\}$, altrimenti è indefinito;
(ii) $Z_\theta[A_1 \vee A_2;B] = Z_\theta[B;A_1 \vee A_2] = Z_\theta[A_1;B] \vee A_2\theta$;
(iii) $Z_\theta[A_1 \wedge A_2;B] = Z_\theta[B;A_1 \wedge A_2] = Z_\theta[A_1;B] \wedge A_2\theta$;
(iv) $Z_\theta[\Box A;\Box B] = \Box(Z_\theta[A;B])$;
(v) $Z_\theta[\Box A;\Diamond B] = Z_\theta[\Diamond B;\Box A] = \Diamond(Z_\theta[A;B])$.

Osserviamo che in generale $Z_\theta[C_1;C_2]$ non è unico, ma dipende sia dalla scelta della coppia di letterali unificabili che possono occorrere in C_1 e C_2, sia dall'ordine di applicazione delle regole (i)-(v).

Una N-formula C è un **risolvente binario** via θ di C_1 e C_2, ed è indicato da $R_\theta[C_1;C_2]$, se esiste $C' = Z_\theta[C_1;C_2]$ definito, tale che C puo' essere ottenuto da C' sostituendo ogni occorrenza di:

$\Diamond\emptyset$	con	$\emptyset$
$\Box\emptyset$	con	$\emptyset$
$\emptyset \wedge B$	con	$\emptyset$
$\emptyset \vee B$	con	B

Notiamo che se C_1 e C_2 sono due clausole della logica classica, allora $R_\theta[C_1;C_2]$ è un risolvente classico di C_1 e C_2.

Consideriamo la formula proposizionale $\Diamond(p \wedge \neg p)$; tale formula è insoddisfacibile e dimostra la necessità di permettere, in logica modale, anche la risoluzione all'interno di una stessa formula. Per questo motivo, definiamo il risolvente unario di una formula, in modo analogo al risolvente binario, con l'aiuto dell'operazione ausiliaria $Z'_\theta[C]$, definita induttivamente come segue.

(i) $Z'_\theta[C_1 \wedge C_2] = Z_\theta[C_1;C_2]$,
oppure: $Z'_\theta[C_1 \wedge C_2] = Z'_\theta[C_1] \wedge C_2\theta$,
oppure: $Z'_\theta[C_1 \wedge C_2] = C_1\theta \wedge Z'_\theta[C_2]$;
(ii) $Z'_\theta[C_1 \vee C_2] = Z'_\theta[C_1] \vee C_2\theta$,
oppure: $Z'_\theta[C_1 \vee C_2] = C_1\theta \vee Z'_\theta[C_2]$;
(iii) $Z'_\theta[\Box C] = \Box(Z'_\theta[C])$;
(iv) $Z'_\theta[\Diamond C] = \Diamond(Z'_\theta[C])$.

Il **risolvente unario** via θ di una N-formula C è definito in modo analogo al risolvente binario.

Il sistema di risoluzione per la teoria Q consiste allora delle regole seguenti:

R1. FATTORIZZAZIONE

$$\frac{C[D \vee F]}{(C[D])\theta}$$

se θ è un upg di D e F

R2. DUPLICAZIONE

$$\frac{C\ [\ D(x^n)\ ]}{C\ [\ D(x^n) \wedge D(y^n)\ ]}$$

se y è una variabile nuova, se x^n occorre soltanto in $D(x^n)$ e se $D(x^n)$ non è nello scopo di più di n operatori modali.

R3. RISOLUZIONE BINARIA

$$\frac{C_1\ ;\ C_2}{R_\theta[C_1;C_2]}$$

se $R_\theta[C_1;C_2]$ è un risolvente binario di C_1 e C_2.

R4. RISOLUZIONE UNARIA

$$\frac{C}{(R'_\theta[C])}$$

se $R'_\theta[C]$ è un risolvente unario di C.

Un insieme S di N-formule è refutabile se e solo se da esso si puo' derivare la formula vuota, $\emptyset$, per mezzo delle regole R1-R4.

La regola R1 corrisponde alla regola di fattorizzazione classica (è qui tenuta distinta dal passo di risoluzione vero e proprio per chiarezza); R3 e R4 costituiscono il nucleo della risoluzione modale, ed R2 corrisponde alla possi-

bilità di riutilizzare una stessa clausola più volte nel corso di una refutazione, cambiando eventualmente nome alle sue variabili. Un semplice esempio (proposizionale) in cui essa risulta necessaria è costituito dalla refutazione dell'insieme $\{ \Diamond \Box p,\ \Box (\Box \neg p \vee \Diamond \neg p) \}$.

$$\dfrac{\dfrac{\dfrac{\Diamond \Box p}{\Diamond (\Box p \wedge \Box p)} \quad ; \quad \Box (\Box \neg p \vee \Diamond \neg p)}{\Diamond (\Box p \wedge \Diamond \neg p)}}{\emptyset}$$

Le restrizioni sulla regola di duplicazione sono necessarie per garantire la coerenza del sistema. Una dimostrazione della coerenza e della completezza del sistema di risoluzione qui presentato si puo' comunque trovare in [4]. Essa utilizza un teorema che stabilisce una "proprietà di Herbrand" per il sistema Q e che esemplifica un metodo relativamente semplice e generale che puo' permettere di ottenere analoghi sistemi di risoluzione generale per altre teorie modali, ivi inclusi i sistemi contenenti la formula di Barcan.

Per esemplificare il procedimento di refutazione, consideriamo l'insieme inconsistente di formule modali costituito da:

$$C_1 = \exists x_1 \Box \forall x_2 \Box (p(x_1) \wedge \neg q(x_1,x_2))$$
$$C_2 = \exists y_1 \Box \forall y_2 (\neg r(y_1,y_2) \vee \forall y_3 \Diamond q(y_3,y_2))$$
$$C_3 = \forall z_1 \Diamond \exists z_2 (r(z_1,z_2) \vee \forall z_3 \Box \neg p(z_3))$$

Come primo passo costruiamo allora l'insieme $T(S) = \{T(C_1),T(C_2),T(C_3)\}$:

$$T(C_1) = \Box \Box (p(c^0) \wedge \neg q(c^0,x_2^1))$$
$$T(C_2) = \Box (\neg r(d^0,y_2^1) \vee \Diamond q(y_3^1,y_2^1))$$
$$T(C_3) = \Diamond (r(z_1^0,f^1(z_1^0)) \vee \Box \neg p(z_3^1))$$

Tale insieme si puo' refutare come segue:

$$\dfrac{\dfrac{\dfrac{\Box \Box (p(c^0) \wedge \neg q(c^0,x_2^1)) \ ; \ \Diamond (r(z_1^0,f^1(z_1^0)) \vee \Box \neg p(z_3^1))}{\{c^0/z_3^1\}}}{\{d^0/z_1^0,f^1(d^0)/y_2^1\}} \quad \dfrac{\Diamond r(z_1^0,f^1(z_1^0)) \ ; \ \Box (\neg r(d^0,y_2^1) \vee \Diamond q(y_3^1,y_2^1))}{\Box \Box (p(c^0) \wedge \neg q(c^0,x_2^1)) \ ; \ \Diamond \Diamond q(y_3^1,f^1(d^0))}}{\{c^0/y_3^1,f^1(d^0)/x_2^1\}}}{\emptyset}$$

Accanto a ciascuna linea di derivazione, abbiamo indicato la sostituzione applicata per la costruzione del risolvente corrispondente.

Come ulteriore esempio, consideriamo l'insieme S seguente:

$$\{ \Diamond (\exists x (\Diamond p(x) \vee \Box \Diamond \forall y \neg q(x,y)) \wedge \Box \forall z \neg p(z) \wedge \Diamond \forall w \exists x' \Box q(w,x'))\}$$

la cui traduzione, T(S), è l'insieme di N-formule:

$$\{ \Diamond ((\Diamond p(c^0) \vee \Box \Diamond \neg q(c^0,y^3)) \wedge \Box \neg p(z^2) \wedge \Diamond \Box q(w^2,f^2(w^2)))\}.$$

Tale insieme è refutato dalla seguente derivazione, che consiste di due applicazioni della regola R3:

$$\dfrac{\dfrac{\Diamond ((\Diamond p(c^1) \vee \Box \Diamond \neg q(c^1,y^3)) \wedge \Box \neg p(z^2) \wedge \Diamond \Box q(w^2,f^2(w^2)))}{\{c^1/z^2\}}}{\dfrac{\Diamond (\Box \Diamond \neg q(c^1,y^3) \wedge \Diamond \Box q(w^2,f^2(w^2)))}{\{c^1/w^2,\ f^2(c^1)/y^3\}}}{\emptyset}$$

CONCLUSIONI

In questo lavoro abbiamo presentato la definizione di un sistema di regole di risoluzione generale per la teoria modale Q, che risulta dalla fusione dei sistemi di risoluzione proposizionale modale già noti con una nozione ristretta di unificazione. Sistemi analoghi possono essere definiti per altre teorie modali, permettendo una generalizzazione del linguaggio di programmazione logica modale, Molog, il quale, attualmente, permette di esprimere soltanto formule in forma prenessa.

Riferimenti bibliografici

[1] Abadi, M., Manna Z., Non clausal temporal deduction, Proceedings of the Logics of Programs Conference, Lecture Notes in Computer Science, Springer-Verlag, 1985.

[2] Aqvist, L., Deontic Logic, in D.Gabbay, F.Guenthner (eds.), Handbook of Philosophical Logic, vol II, D.Reidel Publishing Company, 1984.

[3] Cavalli, A.R., Fariñas del Cerro, L., A Decision Method for Linear Temporal Logic, Proceedings of the 7th International Conference on Automated Deduction, Napa, California, 1984.

[4] Cialdea, M., Une méthode de déduction automatique en logique modale, Tesi di Dottorato, Université Paul Sabatier, Toulouse, 1985.

[5] Combes, P., Leford, V., Gentiane, Un outil de preuve de cohérence de formules en logique temporelle, Note Technique, CNET IPAA ILLC., 1985.

[6] Fariñas del Cerro, L., A Simple Deduction Method for Modal Logic, Information Processing Letters, 14 (2), 1982.

[7] Fariñas del Cerro, L., MOLOG, a System that extends Prolog with Modal Logic, Rapport LSI, Toulouse, 1985.

[8] Fariñas del Cerro, L., Resolution Modal Logic, Logique et Analyse, novembre 1985.

[9] Gallion, D., Implémentation d'un démonstrateur de théorèmes, Rapport LSI, Université Paul Sabatier, Toulouse, 1981.

[10] Hanson, W.H., Semantics for deontic logic, Logique et Analyse 8, 1965.

[11] Henry, J., Implémentation d'un démonstrateur pour les clauses de Horn en logique modale, Rapport LSI, Université Paul Sabatier, Toulouse, 1985.

[12] Herbrand, J., Recherches sur la thèorie de la démonstration, Tesi all'Università di Parigi, 1930, trad. ingl. in W.D.Goldfarb, Logical Writings, D.Reidel Publ. Co., 1971.

[13] Hughes, G.E., Cresswell, M.J., An Introduction to Modal Logic, Methuen, London, 1968.

[14] Lauth, E., Une méthode de résolution linéaire ordonné pour la logique temporelle linéaire, Rapport LSI-ENSEEIHT, Université Paul Sabatier, Toulouse, 1983.

[15] Robinson, J., A Machine Oriented Logic based on the Resolution Principle, J. ACM 12, 1965.

[16] Rychlik, P., The use of Modal Default Reasoning in Information Systems, Institute of Informatics, Warsaw University, 1984.

Definizione di un linguaggio logico per calcolo distribuito.

R. Barbuti*, C. D'Ascanio°, F.Turini*

* Dipartimento di Informatica
Universita' di Pisa
C.so Italia, 40
56100 Pisa

° Selenia S.p.A.
Via S. Maria, 35
56100 Pisa

Abstract

La definizione di un sistema di basi di conoscenza distribuite puo' implicare la definizione di un particolare linguaggio capace di rappresentare la conoscenza e gestire la distribuzione [2,3,5,6,7]. Un approccio interessante sembra il prendere in considerazione un linguaggio particolarmente adatto alla rappresentazione della conoscenza e modificarne opportunamente l'interprete, per estenderlo alle computazioni distribuite.

L'idea centrale di questo lavoro e' appunto quella di partire da un prodotto reale, il PROLOG, e definirne un'estensione [1] che, oltre ad essere orientata alla risoluzione diretta di goals (valutazione diretta), sia eventualmente capace di decomporli in sottoquestioni e di attivare su queste altri processi di valutazione (valutazione di un goal su una teoria), raccogliendone e componendone i risultati (valutazione indiretta).

Lo schema di valutazione risultante e' in sostanza quello di un albero di processi di valutazione in cui i nodi terminali fungono da deduttori logici, mentre quelli intermedi fungono da metadeduttori che possiedono conoscenza circa l'attivita' dei nodi figli (delle teorie su cui essi elaborano, dei goals che valutano e delle modalita' di comunicazione) e ne gestiscono i risultati. La differenza fondamentale tra i due tipi di nodi e' che quelli intermedi sfruttano la capacita' di attivare altri processi di valutazione.

Un nodo puo' attivare vari processi con sottoquestioni differenti sulla stessa teoria, realizzando in tal modo una valutazione parallela (ad esempio per risolvere un goal si possono far risolvere i singoli atomi che lo compongono da processi separati e poi comporne i risultati), ma anche attivare vari processi sulla stessa questione in modo concorrente. La strategia da seguire viene specificata nella teoria associata al nodo di metalivello, ovvero nella teoria a cui fa riferimento il predicato di attivazione.

Le caratteristiche essenziali di una tale estensione sono:
- la capacita' di attivazione di interpreti su teorie (equivalente all'estensione dell'interprete con un predicato predefinito di tipo "demo");
- la capacita' di recupero dei risultati sotto forma di sostituzioni, ed eventualmente anche di valutazioni parziali (la prima parte non e' molto complessa poiche' si tratta di informazioni che il processo interprete restituisce in output, per la seconda parte invece c'e' il non banale problema di decodificare le strutture interne dell'interprete dopo una interruzione);
- uno schema di descrizione sintetico delle informazioni contenute in piu' teorie ed un predicato di selezione su questa per determinare quali teorie dei livello inferiore attivare e su quali goal;
- una tecnica di recupero e di combinazione dei risultati (anche questa come componente della metateoria associata ai metalivelli).

1. Schema

L'idea centrale dello schema proposto riguarda la definizione di metateorie in forma a clausole associate a processi di interpretazione di stile PROLOG. La funzione di queste metateorie e' la definizione, il controllo e la gestione di un calolo distribuito, ovvero esse equivalgono a teorie che lavorano su variabili di tipo *atomo* e *clausola*" e che hanno la possibilita' di attivare valutazioni e recuperarne risultati. Le principali caratteristiche di una simile proposta sono un'elevata flessibilita' (la definizione di un modello distribuito e' interamente contenuta nella metateoria) ed una notevole affinita' con l'interprete PROLOG (utilizzo di uno strumento gia' realizzato). Inoltre un ragionamento di tipo meta e' particolarmente adatto alla realizzazione di un modello distribuito: il nodo di meta-livello funge da collettore (architettura tipo master-slave) per le valutazioni del livello inferiore, permettendo quindi parallelismo e concorrenza e conseguentemente migliorando la velocita' di calcolo rispetto ad una soluzione non distribuita.

Prima di proseguire e' utile sottolineare alcuni concetti di programmazione logica con clausole Horn a cui si fa ampio riferimento nel seguito. Una *valutazione* logica e' l'applicazione di un calcolo inferenziale basato sul principio di risoluzione ad un goal rispetto ad una teoria, ed i suo esito puo' essere di tipo *con successo* se e' possibile ridurre il goal alla clausola vuota, di tipo *con fallimento* altrimenti.
Ad ogni passo di inferenza si tenta di ridurre un singolo atomo del goal attraverso una clausola la cui intestazione sia unificabile con l'atomo stesso: se tale clausola esiste, l'inferenza produce un nuovo goal rimpiazzando l'atomo con il corpo della clausola e applicando la sostituzione prodotta dall'unificazione al risultato; se di tali clausole ne esiste piu' d'una, si producono altrettanti nuovi goal, e per il successo della valutazione e' sufficiente che almeno uno di questi sia riducibile alla clausola vuota.
Il risultato di una valutazione con successo e' la sostituzione *finale* definita dalla composizione delle singole sostituzioni prodotte dalle unificazioni ad ogni passo di inferenza, nell'ordine in cui esse vengono ottenute.
La valutazione *totale* di un goal e' l'insieme di tutte le possibili sostituzioni finali calcolabili, ovvero quelle associate a tutti i possibili modi di ridurre quel goal alla clausola vuota.
Una schematizzazione di questi concetti puo' essere la seguente:

a) La valutazione di un goal g inizia da una coppia

$$< g , \varepsilon >$$

dove ε e' la sostituzione nulla.

b) Un passo di inferenza per resolution applicato ad un goal g con associata la sostituzione θ, rispetto ad una teoria T (con la tecnica depth-first) puo' essere rappresentato come la corrispondenza

$$< g , \theta > \xrightarrow[\text{res T}]{} < g' , \theta' >$$

se esiste una clausola in T tale che e' possibile l'unificazione tra la sua intestazione ed il primo atomo in g con la sostituzione λ.

Allora g' e' il goal $g\lambda$ in cui al posto del primo atomo e' stato riscritto il corpo della clausola e $\theta' = \theta \bullet \lambda$ (con $\bullet$ si identifica l'operazione di composizione tra sostituzioni).
In questo caso si dice che g e' riducibile attraverso T.

c) Una valutazione con successo di un goal g e' espressa da

$$< g , \varepsilon > \xrightarrow[\text{res T}]{}{}^{*} < , \theta >$$

dove con $\xrightarrow{}{}^{*}$ si indica l'esecuzione di un numero finito di passi di resolution.

d) Una valutazione con fallimento di un goal g e' rappresentata da

$$< g , \varepsilon > \xrightarrow[\text{res T}]{}{}^{*} < g' , \theta >$$

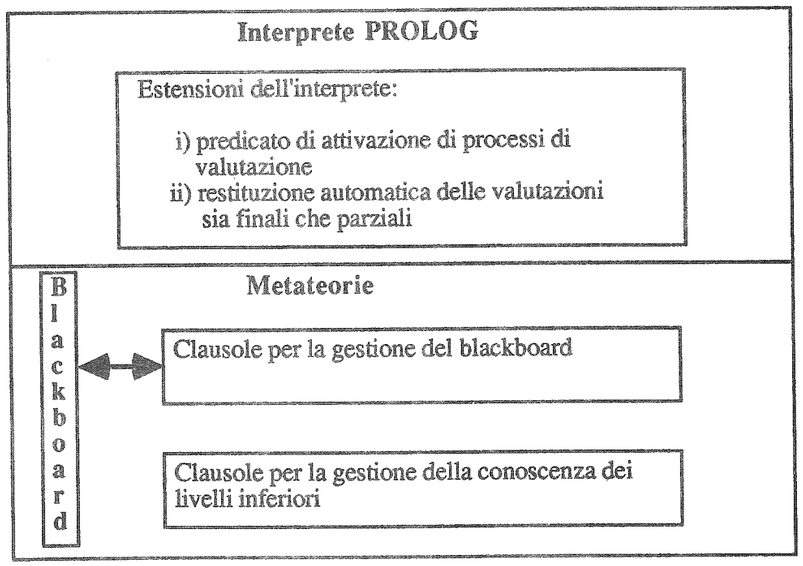
dove g' non e' ulteriormente riducibile attraverso T.

e) Una valutazione totale di un goal g e' un inseme del tipo

$$\{ < , \theta > | < g , \varepsilon > \text{ --->}^* < , \theta > \}$$

In sintesi, una valutazione e' definita da tre elementi: un metodo di calcolo inferenziale (rappresentato da --->), una teoria ed un goal. Possiamo allora definire un *processo di valutazione* in PROLOG come l'attivazione dell'interprete PROLOG (esecutore del calcolo) su un file contenente una teoria in forma a clausole e su di un goal.
Supponendo di disporre di un interprete di PROLOG modificato in modo da permettere l'attivazione di altri processi di valutazione e di un modo per distinguere una teoria da una metateoria, possiamo definire come "processo di valutazione di meta-livello" un processo attivo su una metateoria.

Riallacciandoci anche al discorso iniziale, una possibile rappresentazione dello schema di un processo di meta-livello e' la seguente:



La distinzione tra metateoria e teoria si basa essenzialmente nell'utilizzo del predicato di attivazione e nella definizione di un metodo di recupero dei risultati, al quale ci riferiamo con il termine *blackboard* [4] e' una distinzione puramente indicativa, in quanto ad una qualsiasi teoria capace di manipolare come termini gli stessi costrutti che la definiscono (clausole) e' associabile il concetto di meta.

Una metateoria contiene, sotto forma di clausole, delle conoscenze circa altre teorie, in maniera da essere capace di attivare nuovi processi su quelle stesse teorie. Contiene anche delle clausole per la gestione dei risultati prodotti dai processi.
La presenza di conoscenza circa altre teorie suggerisce una gerarchia: un processo di meta-livello avra' dei processi dipendenti solo sulle teorie note alla sua metateoria.

La struttura dell'attivita' di un simile schema e' un albero di processi di valutazione, in cui i nodi intermedi sono di meta-livello,mentre quelli terminali sono capaci di eseguire valutazioni "dirette" (senza ricorrere al predicato di attivazione).

2. Valutazioni Parziali

Come abbiamo gia' visto, un processo deduttivo si interrompe in due casi: o per una terminazione con successo, oppure per una terminazione con fallimento. In quest'ultimo caso la teoria non contiene sufficienti informazioni per un'ulteriore inferenza per resolution.
Supponendo di lavorare con conoscenze distribuite, e' possibile che in un'altra teoria siano contenute le informazioni sufficienti a "proseguire" il processo di valutazione. e' essenziale allora recuperare anche le valutazioni fallite, considerandole come elaborazioni parziali.
Rifacendoci alla simbologia precedentemente introdotta, una valutazione con fallimento

$$< g , \varepsilon > \text{ --->}^* < g' , \theta >$$
$$\text{res } T$$

viene recuperata proprio nella forma $< g' , \theta >$ e con tale coppia si fa partire un nuovo processo di valutazione su di un'altra teoria T'.
Estendiamo allora le nostre definizioni.
In presenza di piu' teorie $T_1, \ldots , T_n$, $n > 1$, una valutazione *distribuita* con fallimento di un goal g con sostituzione associata θ si ha se e solo se per nessuna teoria g e' ulteriormente riducibile.

Una valutazione *distribuita* con successo di un goal g con associata una sostituzione θ si ha se per un $i \in [1 .. n]$

o

$$< g , \theta > \text{ --->}^* < , \theta' >$$
$$\text{res } T_i$$

oppure

$$< g , \theta > \text{ --->}^* < g' , \theta' >$$
$$\text{res } T_i$$

dove $g \neq g'$

e $< g' , \theta' >$ puo' avere una valutazione *distribuita* con successo per un'altra teoria T_j ,

$$j \in [1 .. n] \quad e \quad j \neq i.$$

Queste definizioni possono essere facilmente comprese riguardo ai processi.
In pratica la necessita' di recuperare valutazioni parziali e' la conseguenza diretta del non possedere (o del non poter mantenere efficientemente) una teoria completa interamente accessibile da un unico processo di valutazione.

Realizzazione dell'Esecutore
Nella presente proposta consideriamo come modello base l'interprete C-PROLOG di [Pereira]. Esso esegue valutazioni di un goal nella maniera depth- first, mantenendo in una pila le possibili riduzioni alternative.Di ogni goal correntemente in esame mantiene la struttura di tipo clausola (*Scheletro*) e l'ambiente (*Sostituzione composta*) .
Quando si ha una valutazione con successo, esso stampa il contenuto dell'ambiente in forma di coppie <nome-della-variabile,termine> su di un file (che di solito e' lo standard-output).
Nel caso di un goal irresolvibile, ne scarta sia lo scheletro che l'ambiente e preleva dalla pila i goals alternativi, se ve ne sono.
Le modifiche da apportare per realizzare un metadeduttore come quello descritto all'inizio sono sostanzialmente:

- l'aggiunta di un nuovo predicato predefinito, *"active"*, che riceve come argomenti una coppia <goal,sostituzione>, il nome di un file contenente una teoria in forma a clausole ed una lista di risultati sotto forma di coppie. Il suo compito e' quello di chiamare l'interprete del linguaggio e far partire un processo di valutazione su quel goal. La presenza di un tale predicato presume anche la possibilita' di far partire un processo di valutazione con un ambiente iniziale non vuoto, ma una tale modifica puo' essere fatta anche nel codice relativo

all'active.
- la possibilita' di comunicare al nodo di livello inferiore affinche' prosegua nella valutazione delle alternative, se ve ne sono (basta consultare lo *stream* con cui il C-PROLOG comunica con l'utente).
- far si che l'interprete restituisca anche i goal parziali (oltre all'ambiente associato) prima di eseguire il *backtracking* .

Nel primo caso si tratta di aggiungere un pezzo di procedura al codice dell'interprete, mentre nel secondo si tratta di porre una routine di stampa nel punto in cui l'interprete ha appurato l'impossibilita' di eseguire riduzioni.

3. Metateoria

E' nella teoria associata al nodo di metalivello che si descrive lo schema distribuito.
Lo scopo dell'attivazione di un processo di interpretazione di metalivello e' sostanzialmente quello di valutare un goal dapprima decomponendolo in sottoquestioni, successivamente attivando i corrispondenti processi ed infine raccogliendone i risultati; su questi risultati viene rieseguita un'ulteriore verifica e se necessario si procede ad un nuovo ciclo di decomposizione-attivazione-raccolta , e cosi' via.
Nel formalismo gia' usato, un singolo passo della valutazione di un nodo di metalivello con teoria MT, contenente informazioni sull'insieme di teorie

$T_1, \dots, T_m$, su un goal g con associata una sostituzione θ , puo' esssere schematizzato come segue:

$$< g , \theta > \text{ ----> } < g' , \theta' > \quad \text{res MT}$$

che equivale ad eseguire i seguenti passi :

(1) $< g , \theta > \text{ ----> } \{ < g_1 , \theta_1 > , \dots , < g_n , \theta_n > \}$ dec MT

dove con dec MT si intende il processo di decomposizione di un goal in sottoquestioni.

(2) $\{< g_1 , \theta_1> , \dots , < g_n , \theta_n > \} \text{ ----> } \{< g_1' , \theta_1' > , \dots , < g_n', \theta_n'> \}$ att MT

dove con att MT si intende l'attivazione di un processo di valutazione per ciascun goal su di un'opportuna teoria, ovvero

$$< g_i , \theta_i > \text{ ----> } < g_i' , \theta_i' > \quad \text{res } T_j$$

per qualche $j \in [1 .. m]$.

(3) $\{ < g_1' , \theta_1' > , .. , < g_n' , \theta_n' > \} \text{ ----> } < g' , \theta' >$ comp MT

dove con comp MT si intende la composizione di una questione su cui eventualmente rieseguire la res MT .

La distribuzione del calcolo si effettua attraverso la dec MT, mentre il controllo e la sintesi dei risultati avviene comp MT.
Puntualizziamo alcuni aspetti. Nella prima figura avevamo evidenziato come la metateoria associata ad un nodo di metalivello fosse composta da due parti : un insieme di clausole per la gestione della

conoscenza dei livelli inferiori ed un insieme di clausole per la consultazione di quello che abbiamo definito blackboard.
La prima di queste contiene le informazioni riguardanti lo *spazio di applicabilita'* di ogni singola sottoteoria , spazio inteso come valutazione della capacita' di risolvere un certo tipo di questioni.
Ad esempio, si puo' dire che una teoria ha la capacita' di (e' *accreditata* a) risolvere un goal se i predicati degli atomi che lo compongono compaiono anche nell'intestazione di qualche clausola della teoria. Ed ancora, supponendo che esistano piu' teorie accreditate, si puo' anche ordinarle in base ad un peso calcolato su certe caratteristiche.
La formulazione di questa parte della metateoria, che d'ora in poi indicheremo con CON (MT) , richiede quindi un metodo per dedurre lo spazio di applicabilita' di una teoria.
Simile e' il discorso riguardante la consultazione dei risultati. Tali risultati si presentano come coppie <goal , sostituzione> dove il goal puo' essere eventualmente la clausola vuota.
Per ognuno di questi si puo' riattivare qualche altro processo, avendo a disposizione un metodo di decisione su quale ordine seguire, o anche si puo' procedere a riformulare una questione da restituire ad un eventuale nodo di livello superiore. L'insieme di clausole che definisce questa sezione sara' indicato con BLACK(MT).
Allora i punti (1), (2) e (3) possono essere ridefiniti come

(i) $< g , \theta > \text{ ----> } \{ (< g_1 , \theta_1 > , T_{s1}) , \dots , (< g_n , \theta_n > , T_{sn}) \}$ res CON(MT)

con $T_{si} \in \{T1 , \dots , T_m \}$, per ogni $i \in [1 .. n]$

(ii) $< g_i , \theta_i > \text{ ----> } < g_i' , \theta_i' >$ res T_{si}

per ogni $i \in [1 .. n]$

(iii) $\{ < g_1' , \theta_1' > , .. , < g_n' , \theta_n' > \} \text{ ----> } < g' , \theta' >$ res BLACK(MT)

Si noti che i passi di valutazione di CON(MT) e BLACK(MT) spazzano su insiemi di coppie < goal , sostituzione >.

4. Esempio ed ulteriori considerazioni.

Nella forma stessa delle clausole Horn e' insito il concetto di *serializzazione* del calcolo attraverso passi di valutazione parziale.
Un goal della forma

$\text{<---- } A_1 , A_2 , \dots , A_n$

viene risolto dall'interprete PROLOG, basato sul principio di risoluzione, a partire dal primo atomo della sentenza.
In pratica la valutazione si sviluppa risolvendo un goal del tipo

$\text{<---- } A_1$

ricavandone , in caso di successo, una sostituzione $\theta 1$ e reiterando il procedimento su

$\text{<---- } A_2\theta_1$

e cosi' via.
Se ad ogni scalino di questa cascata di valutazioni si mantiene la sostituzione prodotta, alla fine del procedimento la loro composizione (secondo l'ordine) costituira' il risultato della valutazione.
La serializzazione corrisponde alla trasmissione di valutazioni di alcune variabili *condivise* da piu' atomi, ma concettualmente la resolution sugli atomi si potrebbe effettuare anche a guisa di processi comunicanti: quando un processo termina , trasmette la sostituzione calcolata agli altri processi, che la incorporano (la *applicano* ai propri goal) verificandone anche la consistenza con i loro risultati precedenti.
Riportando questo discorso in un ambiente di informazioni distribuite, ogni atomo puo' essere

valutato rispetto ad una teoria che si suppone abbia sufficienti conoscenze per farlo.
Un semplice esempio di metateoria destinata a tale scopo puo' essere il seguente insieme di clausole:

```
demo ([]) <---- ;

demo ([[[]|subst]|list-of-pair]) <----
                                    write (subst),
                                    demo (list-of-pair);

demo ([[[A|list-of-atom]|subst]|list-of-pair]) <----
                                    pred-of (A,P),
                                    search (P,T),
                                    active ([A|subst],T,list-of-pair1),
                                    comp(list-of-pair1,list-of-atom,new-list),
                                    demo (new-list),
                                    demo (list-of-pair);
```

dove
- *pred-of* seleziona il predicato P dell'atomo A,
- *search* e' il predicato di ricerca della teoria piu' accreditata a risolvere questioni inerenti un predicato,
- *active* fa partire un processo di valutazione del goal <--- *A* con sostituzione associata *subst* sulla teoria contenuta in T e restituisce tutte le valutazioni (parziali e finali) nella lista *list-of-pair1*.
- *comp* riceve come argomenti due liste e costruisce una lista di liste aventi come testa un elemento del primo argomento e come coda il secondo argomento.
- *write* e' un predicato predicato predefinito per la stampa delle sostituzioni risultato
- *demo* rappresenta l'esecutore.

Oltre a queste clausole la metateoria conterra' le informazioni circa il predicato di ricerca e quello di composizione.Si noti che manca la verifica per l'interrruzione del procedimento in caso di non esistenza di valutazioni con successo Per ribadire i concetti espressi in precedenza, possiamo dire che in questa metateoria si evidenzia solo la gestione della conoscenza dei livelli inferiori, mentre la distribuzione del calcolo e' effettuata dalla *demo* e il blackboard e' costituito dalle liste che sono argomenti dei predicati *demo* e *active* .
Ulteriore lavoro sara' svolto per rappresentare schemi piu' complessi che descrivano meglio le possibilita' offerte dalla nostra proposta.
Un non banale argomento di speculazione e' , ad esempio, quello di gestire l'acquisizione di nuove conoscenze a partire da quelle gia' contenute nelle teorie, che comprende anche problemi di verifica e mantenimento della consistenza di nuove teorie.

Bibliografia

[1] Bowen K.A. e R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. *Logic Programming*, (Clark K.L. e Tarnlund S.A. Eds), Academic Press, 1982.

[2] Corkill D.D. e V.R. Lesser. Functionally Accurate, Cooperative Distributed Systems. *IEEE trans. on System, Man and Cybernetics vol. SMC-11*, 1, 1981.

[3] Corkill D.D. e V.R. Lesser. The Use of Metalevel Control for Coordination in a Distributed Problem Solving Network. IJCAI, Karlsruhe, 1983.

[4] Ermann L.D., F. Hayes-Roth, V.R. Lesser e D.R. Reddy. The HEARSAy-II Speach Understanding System: Integrating Knowledge to Resolve Uncertainity. *Comp. Surveys, vol 12*, 2, 1980

[5] Filmann R.E. e D.P. Friedman. *Coordinated Computing*,

[6] Fukurawa K. et al. Mandala: a Logic Based Knowledge Programming System. Int. Conf. on Fifth Generation Computer Systems, Tokyo, 1984.

[7] Hewitt C.E. e W.A. Kornfeld. The Scientific Community Metaphor. *EEE trans. on System, Man and Cybernetics vol. SMC-11*, 1, 1981.

UN'ESTENSIONE DI PROLOG PER UN AMBIENTE ROBOTICO

A. De Santis *, A. Guercio*, S. Levialdi°, G. Tortora*

* Dipartimento di Informatica ed Applicazioni, Università di Salerno
° Dipartimento di Matematica, Università di Roma

ABSTRACT

Per poter operare in modo sicuro ed efficiente in real time, bisogna poter manipolare evoluzioni inattese di una scena. A tale scopo occorre una continua interazione tra il robot ed il suo ambiente, interazione che comporta un costante monitoraggio per rilevare evoluzioni della scena ed inoltre, la possibilità di prendere decisioni intelligenti. Per un tale modello dinamico, un ambiente Prolog-like, sembra essere particolarmente adatto, grazie alle sue caratteristiche task/oriented.

Le limitazioni dell'ambiente Prolog non vengono trascurate ed allo scopo di superarle viene proposta una estensione di Prolog (Prolog-RL).

INTRODUZIONE

In un ambiente robotico nel quale cooperano un manipolatore meccanico ed una telecamera allo scopo di compiere un ben definito task (che può cambiare di volta in volta), il software ed il linguaggio di programmazione giocano un ruolo cruciale nella progettazione e nell'implementazione di strategie di controllo. Per fornire convenienti strumenti software per supportare programmi di scrittura, di test e di debugging che devono cooperare in un ambiente robotico, sono stati proposti molti suggerimenti spaziando dalla Programmazione Robotica all'Intelligenza Artificiale.

In un precedente articolo[1] era stata proposta una estensione dell'interprete Prolog per gestire gli inputs da un sensore per la visione (es. una telecamera) ed un manipolatore (es. un occhio meccanico) in modo da costruire sperimentalmente una base di dati originata dagli input sensori ed in seguito, un framework concettuale, dedotto dalla precedente base di dati. La prima base di dati è stata chiamata Base di Dati Percettiva (BDP) mentre la seconda é stata chiamata Base di Dati Concettuale (BDC) e può essere considerata equivalente alla base della conoscenza correntemente usata nella tecnologia dei sistemi esperti.

Le informazioni collezionate dai sensori, in molte applicazioni pratiche, non sono sufficienti per la completa progettazione ed esecuzione del task, di conseguenza potrebbe essere necessario recuperare e dedurre nuovi dati dalla Base di Dati Concettuale. Se una particolare situazione lo richiede, questa base di dati può essere espansa collezionando dall'ambiente nuovi dati; i sensori saranno quindi orientati ad una acquisizione dell'informazione la più completa possibile. Per ottenere le facilities attese, abbiamo esteso il Prolog esistente per mezzo di due predicati built-in: uno che consente la ricerca della prossima azione eseguibile compatibilmente al contesto corrente (es. per un particolare scenario che è presente durante l'attività) l'altro progettato per l'esecuzione pratica di azioni consentite.

Nella prossima sezione descriveremo questa estensione ed infine includeremo un esempio pratico della sua utilizzazione.

ACQUISIZIONE DELL'INFORMAZIONE

Quando desideriamo effettuare un particolare task usando un sistema costituito da un braccio manipolatore ed una telecamera, dobbiamo considerare un gran numero di azioni gerarchiche, ricorsivamente, fino a raggiungere un'azione di base, che é rappresentata su inputs primari chiamati 'atomi'. Questa catena di eventi può essere rappresentata nel seguente modo:

ed è ottenuta da un'analisi del task in accordo all'attuale situazione come rilevata dai sensori. Quando questa situazione cambia, allora la catena degli eventi può essere modificata in accordo alla situazione.

Riferendoci alla figura 1, possiamo vedere che i sensori artificiali forniranno segnali che sono una versione campionata della scena attraverso inputs tattili e visuali. Tali segnali devono essere connessi in modo tale da rendere utilizzabile questa informazione per poi traslarla in un struttura di fatti Prolog [2,3]. Ogni caratteristica rilevata sarà trasformata in fatti Prolog per mezzo di un programma speciale; l'insieme di tutti i fatti registrati costituisce la base dell'informazione percettiva direttamente acquisita dai sensori.

Una scena esterna può essere descritta da un operatore umano specializzato, in termini di concetti base che descrivono e modellano i principi di base ed i meccanismi coinvolti nella performance del task. Il contributo di una tale conoscenza concettuale è riflesso nei predicati built-in della Base di Dati Concettuale che evolverà grazie a due meccanismi:

1) quando nuove situazioni devono essere risolte (evoluzione di una scena)
2) quando si effettuano inferenze da parte dell'uomo (intervento esterno).

Il secondo meccanismo può, in futuro, essere sostituito da un sistema esperto.

Poiché l'ambiente software è Prolog oriented, la base di dati sarà scritta come un programma Prolog-like. Il sinergismo tra la Base di Dati Concettuale ed la Base di Dati Percettiva produrrà i segnali di controllo che abiliteranno il sistema ad eseguire il task in accordo agli ultimi segnali input ricevuti e l'inferenza concettuale sarà aggiornata sulla base dei suggerimenti umani. Poichè la scena potrebbe cambiare nel tempo, dovrebbe essere fornito un altro meccanismo che potrebbe o fermare la corrente esecuzione, o, in una versione più sofisticata, riconoscere le differenze, durante l'evoluzione di una scena, abilitando una nuova strategia.

L'ESTENSIONE PROLOG

L'estensione suggerita deriva dalla necessità di:

a) aderire ad un ambiente robotico e perciò arricchire il linguaggio con operazioni naturali per detti tasks, es. azioni;
b) far uso e giovamento dei meccanismi di inferenza per convalidare solo azioni compatibili;
c) abilitare l'esecuzione di quelle azioni che sono compatibili e derivano dall'ultima "istantanea" presa dalla scena. Noi descriveremo, ora, ognuna di queste estensioni suggerite

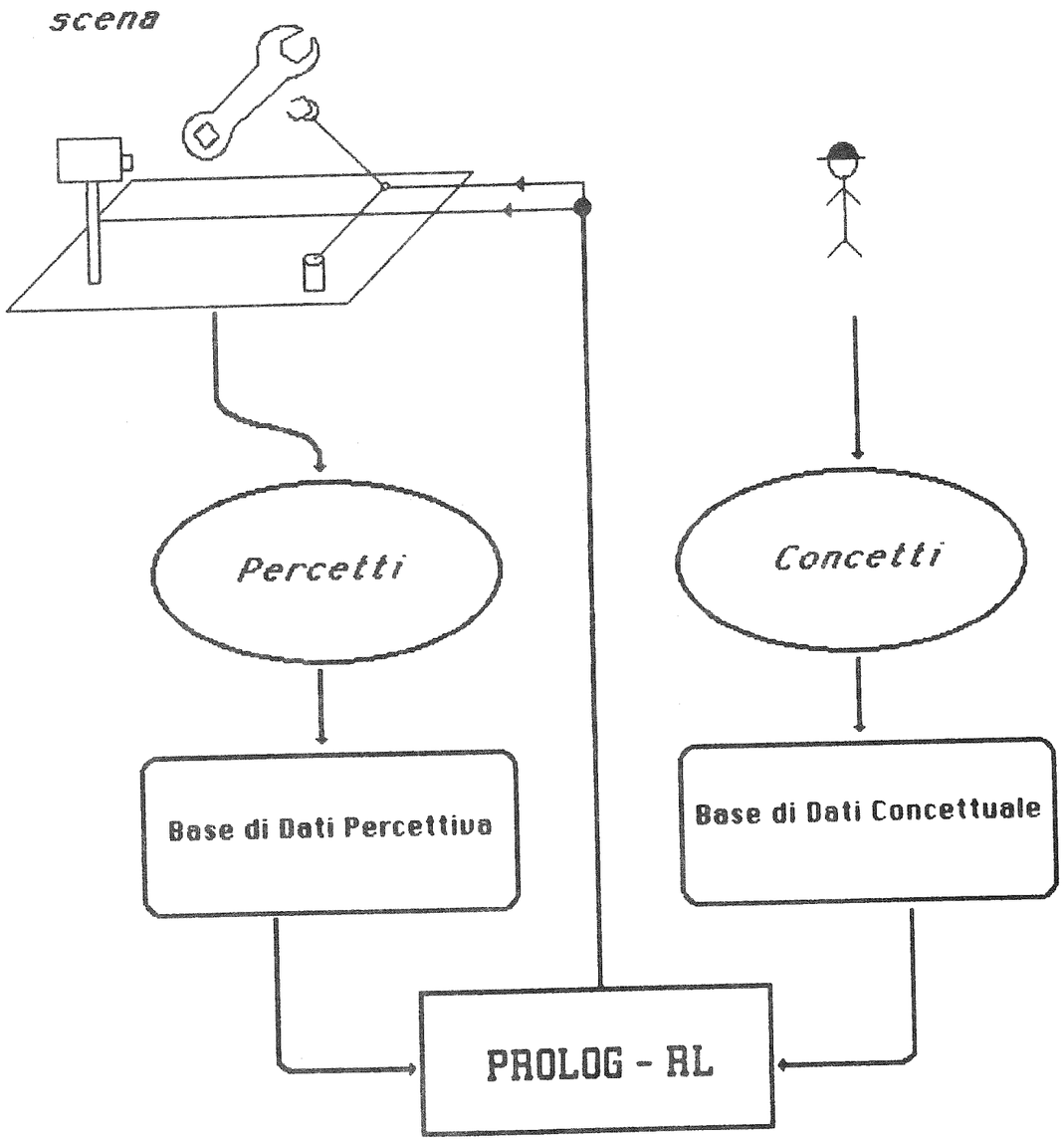


fig. 1. I sensori forniranno segnali che sono una versione campionata della scena.

per una piena comprensione del nostro approccio.

Introduciamo, prima di tutto, la definizione di 'azione', ed osserviamo che le azioni possono rappresentare o un'operazione elementare o una sequenza di operazioni eventualmente inclusa in una espressione. Possiamo adoperare la seguente definizione ricorsiva

1. (A_1)
2. $A_1 \hat{} A_2$
3. A_1 , A_2

sono azioni, dove A_1 ed A_2 sono azioni elementari, " ^ " è un AND-parallelo logico e " , " è un AND-sequenziale logico. Finalmente, una qualsiasi combinazione di 1,2 e 3 è anche un'azione.

Una regola Prolog-RL ha due componenti, una testa seguita da un corpo, separata dal simbolo :-. La testa conterrà o un fatto o un'azione ed il corpo, come in Prolog standard, una sequenza di fatti. Ogni qualvolta un'azione è presente nella testa di una regola, Prolog-RL proverà a soddisfare i sottogoals contenuti nel corpo facendo backtrack finchè tutti i sottogoals sono stati analizzati. A mo' di semplice esempio, considereremo l'implementazione delle azioni **ROTATE** e **MOVE_FAST** (nella testa) nel caso in cui un animale (un elefante) è vicino al robot (nel corpo).

La seguente regola

ROTATE(180°), **MOVE_FAST** :- **elephant**(X), **range**(X,close).

rappresenta le azioni combinate di rotazione e di movimento ogni volta che un elefante esiste e la sua distanza dal robot è al di sotto di una soglia (close).

La seconda estensione è l'aggiunta di un predicato built-in **next_action(L)**, dove L è la lista delle espressioni che rappresenta la prossima azione eseguibile (che non deve essere necessariamente eseguita).

Prolog-RL cercherà la prima clause che sarà soddisfatta nella Base di Dati Concettuale restituendo in output la testa della clause ed assegnandola ad L.

Inoltre questo predicato è un conveniente strumento di test e di debugging per grossi programmi.

La terza ed ultima estensione consiste in un secondo predicato built_in chiamato **activate**, che permetterà l'esecuzione della prossima azione eseguibile sulla base della valutazione della precedente azione e tenendo, contemporaneamente, conto dello stato della scena. Per attivare un' azione affianchiamo al punto interrogativo (prompt del sistema) il predicato **activate**; ciò è completamente equivalente a mandare in esecuzione la prossima azione eseguibile sulla base della valutazione della presedente azione.

Vale a dire che

?- **activate**.

è equivalente a

?- **b**.

dove b è l'output del predicato **next_action(L)**.

ESEMPIO

Per chiarire la meccanica di Prolog-RL, descriveremo uno scenario (simile a quello dato da Maletz[4]) utile più da un punto di vista pedagogico che non reale.

La scena contiene i seguenti oggetti: uomini, papere, topi ed un elefante che saranno osservati da

una telecamera e seguiti (o meno) da un braccio meccanico posto su una piattaforma di locomozione. Il comportamento del robot richiesto è di ignorare gli uomini e le papere, scappare dall'elefante (se è troppo vicino) ed inseguire i topi.

Le caratteristiche della scena possono essere estratte grazie alle devices di acquisizione e combinate logicamente in fatti Prolog nel seguente modo: la grandezza di un oggetto X, può essere descritta col fatto **size**(X,K) dove K può assumere valore large o small, il numero di gambe di un oggetto, con **legs**(X,L) dove L può assumere valore 2 o 4, la distanza dell'oggetto dal centro della scena, con **range**(X,Y) dove Y può assumere valore 'close' o 'far'.

Tutti i dati precedenti costituiscono la Base di Dati Percettiva al quale deve essere integrato la Base di Dati Concettuale.

```
human(X) :- legs(X,2) , size(X,large).
duck(X) :- legs(X,2) , size(X,small).
rat(X) :- legs(X,4) , size(X,small).
elephant(X) :- legs(X,4) , size(X,large).
```

In accordo al comportamento richiesto saranno effettuate le seguenti azioni:

HALT ferma il movimento del braccio
ROTATE(30°) ruota la telecamera di un angolo specifico
MOVE_FAST sposta rapidamente la posizione del braccio verso il centro della scena o lontano da essa (nota che il centro della scena è il target della telecamera)
ZOOM_IN posiziona il braccio in corrispondenza dell'oggetto visto dalla telecamera in un dato istante.

Il programma Prolog_RL che soddisfa le precedenti richieste contiene le seguenti regole:

a) **ROTATE**(alfa) :- **human**(X); **duck**(X).
(Se la telecamera è puntata verso un uomo o una papera, ruotala di alfa gradi)

b) **HALT** ^ **ROTATE**(alfa) :- **elephant**(X), **range**(X,far).
(Se la telecamera individua un elefante e la sua posizione è lontana allora il braccio viene fermato e la telecamera ruota di un angolo alfa)

c) **RO TATE**(180°) , **MOVE_FAST** :- **elephant**(X), **range**(X,close).
(Se c'è un elefante nella scena ed è vicino, la telecamera ruota di 180° ed il braccio è mosso rapidamente verso la posizione puntata dalla telecamera)

d) **ZOOM_IN**(X), **MOVE_FAST** :- **rat**(X).
(Se l'oggetto individuato è un topo, allora il braccio lo seguirà).

Supponiamo che in una scena, in un dato istante, siano presenti un elefante ed un topo. La Base di Dati Percettiva (BDP) sarà aggiornata con i seguenti fatti (ricavati dai sensori):

```
1) range(1,far).
2) legs(1,4).
3) size(1,large).
4) range(2,far).
5) legs(2,4).
6) size(2,small).
```

Per esaminare quale azione è possibile effettuare, attiviamo il predicato **next_action(L)**. A partire dalla prima clause nella Base di Dati Concettuale (BDC), saranno analizzate solo le regole che hanno una azione nella testa. Se il match ha successo, Prolog_RL costruirà una lista contenente tutti i comandi elementari richiesti per l'esecuzione dell'azione. La prima clause, a), fallirà; infatti non è possibile che un match sia soddisfatto visto che nella scena non c'è né un uomo né una papera, (ogni sottogoal chiama la sua specifica nel BDC per testare la presenza dell'oggetto uomo o papera). Sarà ora analizzata la seconda clause b), e questa volta il match è soddisfatto rendendo attivabile

l'azione contenuta nella testa. L'esecuzione di tale azione sarà agganciata solo dal predicato **activate**. Proseguendo, la clause c) non sarà soddisfatta mentre la clause d) termina con successo.

Come abbiamo visto, Prolog è stato esteso tenendo presente i vincoli e le richieste di un ambiente robotico, sfruttando la sua capacità di dialogare interattivamente con l'utente e di aggiornare con facilità la sua base di dati ogni qualvolta si ritiene necessario, coinvolgendo in tal modo la scelta delle azioni future nella scena.

BIBLIOGRAFIA

1. M. De Blasi, A. Guercio, S. Levialdi, G. Tortora, "Prolog-RL: a Proposal for a Task-oriented Language", MIMI 84, Int. Conf. on Mini and Micro Computers and their Applications, Bari, 1984, pp. 56-58.

2. W. F. Clocksin, C. S. Mellish, "Programming in Prolog", Springer-Verlag, Berlin, 1981.

3. K. L. Clark, S.A. Tarnlund, edits. "Logic Programming", APIC Studies in Data Processing, Academic Press, New York, 1982.

4. M. C. Maletz, "An Introduction to Multi-Robot Control using Production Systems", Proc. IEEE Symp. on Languages for Automation, Chicago, 1983, pp. 22-27.

PROGRAMMAZIONE LOGICA MODULARE IN AMBIENTE LISP

F. Bergadano

Dipartimento di Informatica, Universita` di Torino
Via V. Caluso 37, 10125 - Torino

ABSTRACT

In questo lavoro e` descritta una estensione del linguaggio di programmazione Loglisp, definito nella sua forma piu` semplice da Robinson e Sibert. La principale innovazione consiste nella possibilita` di suddividere la conoscenza in moduli separati e interagenti (contesti). E` stato inoltre introdotto un insieme di primitive che permettono di modificare, estendere o cambiare l'insieme di regole attivo, anche nel corso di una stessa deduzione. Tali possibilita`, unite alla fusione tra Lisp e logica, che e` alla base della stessa versione originaria del linguaggio, assicurano uno strumento maneggevole e potente.

1. INTRODUZIONE

La logica del primo ordine e` stata tradizionalmente impiegata come strumento di descrizione, utile per formalizzare concetti e guidare il ragionamento. Un'interpretazione che pone invece in primo piano l'aspetto procedurale della logica e` stata proposta da Kowalski [1]: in quest' ottica, un insieme di asserzioni del primo ordine e` visto non solo come descrizione di una situazione (aspetto dichiarativo), ma anche e soprattutto come specifica formale di una serie di possibili inferenze. La programmazione logica, che si basa su questo approccio, e` risultata particolarmente efficace nell'affrontare problemi di natura non deterministica, la cui soluzione va ricercata percorrendo diversi possibili cammini. In questi casi il programmatore fornisce un "metodo di soluzione" senza esplicitare la serie di operazioni da eseguire. Tuttavia, quando la via da seguire e` nota, allora e` piu` naturale per l'uomo e piu` efficiente per la macchina precisare la procedura che risolve il problema.

Si consideri ad esempio il problema di decidere se due liste di predicati binari sono uguali a meno di una rinominazione degli argomenti.

Formalmente, siano date le due liste:

$$a = ((A_1 \ x_{i_1} \ x_{j_1}) \ldots (A_n \ x_{i_n} \ x_{j_n})) \qquad (1)$$

$$b = ((A_1 \ y_{i_1} \ y_{j_1}) \ldots (A_n \ y_{i_n} \ y_{j_n}))$$

Nelle (1) $t \neq u$ non implica necessariamente $A_t \neq A_u$, $i_t \neq i_u$, $j_t \neq j_u$. Si vuole

sapere se esiste una corrispondenza che associa ad ogni variabile y_t occorrente in b una variabile x_s occorrente in a, che renda uguali le due liste. Tale problema puo` essere risolto in un linguaggio di programmazione logica con la semplice regola(+) :

```
uguale(a,b) <- "asserisci a",
               b,
               "cancella le asserzioni fatte",
               "asserisci b",
               a.
```

Lo stesso problema richiederebbe un maggior sforzo di programmazione se affrontato con uno stile procedurale. Infatti un programma scritto per questo scopo dovrebbe essenzialmente implementare un algoritmo esaustivo, capace di tentare una delle varie associazioni tra variabili e cambiare alternativa quando questa risulti impossibile da portare avanti. Il programmatore dovrebbe preoccuparsi di tutti i dettagli implementativi che questo richiede e potrebbe piu` facilmente incorrere in un errore.

Se invece il problema si presta ad essere risolto in modo deterministico, ovvero se si e` trovato un metodo di soluzione che non sia una mera ricerca tra diverse possibili vie, ma una esplicita sequenza di passi da eseguire, la programmazione logica puo` risultare scomoda o artificiosa. L'antitesi tra conoscenza dichiarativa e procedurale si risolve nella sintesi che suggerisce di utilizzarle entrambe in modo integrato, preferendo l'una o l'altra in base al caso in esame.

La necessita` di questa sintesi costituisce il motivo che e` alla base della creazione e dell'utilizzo del linguaggio Loglisp, che permette di utilizzare nello stesso programma regole di tipo Prolog e funzioni Lisp. Nel sistema di logica un predicato puo`, anziche` essere dimostrato, venire valutato mediante la funzione ad esso associata. Viceversa il programma Lisp puo` richiedere la dimostrazione di alcuni predicati a partire da un certo insieme di regole. In secondo luogo questo lavoro si propone di descrivere un metodo per strutturare in moduli separati i programmi scritti in Loglisp. Questa e` in effetti una possibilita` utile per l'implementazione dei sistemi di Intelligenza Artificiale, per i quali in particolare si vuole proporre il linguaggio. Infatti in tali sistemi e` richiesto che la conoscenza usata, il motore inferenziale e il controllo siano il piu` possibile espliciti ed espressi in forma chiara. Uno stile di programmazione logica che permetta di strutturare la conoscenza e di utilizzare funzioni scritte in Lisp puo` facilmente soddisfare queste richieste.

Nel Paragrafo 2 si spiega il meccanismo di deduzione e il metodo usato per fondere Lisp e logica in un unico linguaggio. I contesti e le primitive che operano su di essi sono introdotti nel Paragrafo 3; il Paragrafo 4 contiene un esempio di programmazione in questo linguaggio.

2. IL SISTEMA DI DEDUZIONE

Loglisp non fa uso, come Prolog, di un meccanismo di "backtrack", ma mantiene tutta l'informazione utile dell'albero di deduzione. Un nodo dell'albero contiene una coppia (Q E), dove Q e' una lista di predicati ancora

---

(+) Dopo la descrizione dei contesti e delle operazioni che possono essere effettuate su di essi (Par. 3), rimarra` chiaro come si possano scrivere in Loglisp le frasi tra virgolette.

da dimostrare ed E e` una lista di coppie variabile-valore che specifica i legami delle variabili in Q. Un nodo tale che Q=nil e` una soluzione del problema. I figli di un nodo (P.Q E) sono tutti i nodi (A.Q E') tali che P' <= A appartiene all'insieme di regole dato e P e P' sono unificabili in E con risultato E'. Per un'esposizione piu` particolareggiata di come venga gestito e implementato questo procedimento si rimanda a Robinson e Sibert [2]. L'algoritmo di deduzione, essenzialmente descritto in Fig. 1, mantiene una lista OPEN che comprende tutte le foglie dell'albero che non sono soluzione.

```
Soluzioni <- ∅
OPEN <- ∅
while OPEN ≠ nil and |Soluzioni| < numr
        1. scegli un nodo (Q E) da OPEN
        2. Q' <- (Lisp-semplifica Q E)
        3. if Q'=nil then Soluzioni <= Soluzioni ∪ E
                     else "espandi (Q' E)"
                          "aggiungi a OPEN i nodi ottenuti"
```

Fig.1 Algoritmo di deduzione.
La variabile numr indica il numero di soluzioni richieste.

Il passo 2 garantisce la fusione con l'ambiente Lisp ed e` descritto piu` in dettaglio nelle Fig. 2 e 3. La funzione Lisp-valuta ha come argomento una s-expression L e restituisce una coppia di valori (L' esito); esito vale "t" se e` stato possibile valutare completamente L in Lisp, "errore" altrimenti. L' contiene la s-expression ottenuta valutando L. Come risulta dall' algoritmo e` possibile che tale valutazione sia solo parziale. Se ad esempio L e` una lista il cui primo elemento non corrisponde a un identificatore di funzione, non e` ovviamente possibile effettuare la valutazione e ottenere un nuovo valore. Tuttavia, se altri elementi di L sono correttamente valutabili in Lisp, il loro valore verra` sostituito in L' nella posizione corrispondente. La funzione Lisp-semplifica, che ha come argomento una lista di predicati Q e una lista E che specifica i legami delle variabili in Q, restituisce una nuova lista di predicati Q'. L'algoritmo, descritto in dettaglio in Fig. 2, applica la funzione Lisp-valuta a ogni elemento di Q e si arresta quando il valore restituito e` nil o quando non e` piu` possibile effettuare una valutazione completa. Nel primo caso un predicato e` stato giustamente valutato in Lisp ma e` risultato falso, e il ramo corrente della deduzione deve essere troncato. Nel secondo caso non e` stato possibile eseguire un predicato come procedura, e sara` quindi necessario dimostrarlo con le regole date, effettuando un altro passo dell'algoritmo principale descritto in Fig. 1.

In questo modo e` possibile eliminare completamente la distinzione sintattica tra funzioni Lisp e predicati logici, che ancora esiste nella versione di Robinson e in alcune versioni commerciali; un'espressione sara` considerata predicato solo nel caso che contenga variabili non legate o che la sua valutazione Lisp risulti impossibile. Uno stesso simbolo puo` cosi` essere utilizzato in entrambi i modi, come dimostra il semplice esempio riportato nel seguito. Si definisca la funzione "div":

```
(defun div (x y z) (equal (quotient x y) z))
```

e si aggiunga all'insieme di asserzioni logiche il fatto

```
(div x 0 'infinito)
```

Quando si chieda al sistema se (div 8 4 2) sia vero, questo ne effettuera` la valutazione, mentre (div 8 0 'infinito) sara` trattato come predicato. In entrambi i casi la risposta sara` affermativa. In Loglisp e` cosi` risolto

in modo del tutto naturale il rapporto tra dimostrazione formale e interpretazione o, se vogliamo, tra conoscenza procedurale e conoscenza dichiarativa. Cio` che e` per sua natura una procedura e` codificato in un programma Lisp e valutato al momento opportuno della deduzione, mentre quello che si formalizza bene in enunciati del primo ordine e` scritto nel programma Loglisp sotto forma di regole tipo-Prolog. Si puo` cosi` avere uno scambio di informazione in due direzioni: un programma Lisp puo` richiedere alla logica la dimostrazione di un teorema e la logica puo` chiedere al Lisp la valutazione di un predicato o di una funzione. Anzi e` possibile muoversi in entrambe le direzioni ottenendo un annidamento a piu` livelli tra i due ambienti, anche in modo ricorsivo.

Lisp-semplifica Q E

```
R  <- (car Q)
P  <= E(R)
P' <- (lisp-valuta P)
if P' = (P'' errore)      then Q' <- (P'' (cdr Q))
if P' = (nil t)           then Q' <- fail
if P' = (s-expression t)  then Q' <- (lisp-semplifica (cdr Q))
```

Fig. 2. Algoritmo per semplificare mediante una valutazione Lisp una lista di predicati.

Lisp-valuta L

```
if "L e` un atomo" then if L e` unbound then (return (L errore))
                                         else (return ((eval L) t))
; sia L = (P x1 ... xn)
{x'i} <- (Lisp-valuta xi)
L'    <- (P x'1 ... x'n)
if (∃ i) x'i = (y,errore)  then (return (L' errore))
if "(eval L')) ritorna senza errore y"  then (return (y t))
                                        else (return (L' errore))
```

Fig. 3. Algoritmo per valutare in Lisp un predicato.

Nella presente implementazione e' stata anche introdotta la possibilita` di avere la traccia della deduzione. Nel caso che l'utente abbia attivato l'opzione "tracing", il sistema aggiunge a ogni nodo dell'albero presente in OPEN informazioni sulla storia della deduzione che ha permesso di giungervi. In tal modo il sistema e` in grado di ricostruire la serie di regole applicate per giungere alle soluzioni. Questo "tracing" e` piu` selettivo di quello, peraltro utilizzabile, del Lisp, perche` non menziona le deduzioni che non hanno portato ad alcuna soluzione. Se al contrario la traccia esplicitasse tutti i passi eseguiti dal sistema, anche nel caso di "backtracking", l'utente potrebbe essere costretto ad analizzare una grande quantita di informazione inutile. Questo, come altri vantaggi, sono pagati dal Loglisp in termini di efficienza.

3. I CONTESTI

Le regole che costituiscono la base di conoscenza di cui si fa uso per eseguire le deduzioni richieste sono memorizzate in una "property" dei predicati che sono conseguenti delle regole stesse. Di fatto ogni predicato puo` avere

piu` di una "property", permettendo cosi` di avere piu` di un insieme di regole disponibile per il sistema. In questo modo sara` possibile non solo dedurre teoremi inferibili da diverse premesse, ma anche cambiare o modificare la base di conoscenza nel corso stesso della deduzione. Ecco la descrizione formale di come vengono organizzati i dati:

Per contesto si intende un insieme di regole della forma:

$(P\ x_1 \ldots x_n) \leftarrow (A_1\ y_1 \ldots y_{k_1})$

$\ldots$

$(A_m\ z_1 \ldots z_{k_m})$

Si supponga che nel sistema siano presenti n contesti $C_1, \ldots, C_n$. Siano inoltre $P_1, \ldots, P_m$ i predicati che figurano come conseguente di almeno una regola di almeno un contesto. Sia $R_{i,k}$ l'insieme delle regole del contesto C_k aventi come conseguente il predicato P_i. Ogni predicato P_i avra`, per ogni contesto $C_k$ cui appartenga almeno una regola di cui $P_i$ e` il conseguente, una "property" con chiave C_k e valore $R_{i,k}$. In termini Lisp, $R_{i,k}$ = (get P_i C_k). Si ha inoltre un contesto, denominato contesto attivo, le cui regole vengono impiegate nel ciclo di deduzione. Sono infine date le seguenti primitive di sistema, che permettono di modificare e gestire i contesti:

← (use C)	usa C come contesto attivo
← (copy C_1 C_2)	copia il contesto C_1 in C_2
← (add C_1 C_2)	aggiunge tutte le regole di C_1 a C_2
← (assert regola)	aggiunge regola al contesto attivo
← (stack C)	aggiunge il nome di contesto C allo stack
← (unstack)	preleva dallo stack il primo contesto e lo usa come contesto attivo.

Queste primitive richiedono soltanto operazioni sui puntatori, non accedendo direttamente alla struttura dei contesti o delle regole, e sono pertanto efficienti.

In questo sistema, come in Prolog, la negazione non puo` che essere interpretata come fallimento, anche se in linea di principio dire che un predicato non e` vero non coincide con l'impossibilita` di dimostrarlo. Tuttavia l'utilizzo dei contesti permette di implementare in modo corretto l'implicazione, qualora l'antecedente non sia contraddittorio. Siano ad esempio B un enunciato del primo ordine, C_B il contesto che contiene le regole tipo-Prolog corrispondenti e Q una lista di predicati. In questa versione di Loglisp e` possibile verificare se B implica Q con la regola

(implica B Q) <- (stack), (use C_B), Q, (unstack) -

mentre senza far uso di contesti o meccanismi che permettano di asserire e cancellare parte delle regole usate in modo efficiente si avrebbe una soluzione del tipo

(implica B Q) <- (not B) -

(implica B Q) <- Q ←

che non e` equivalente e discorda con il concetto classico di implicazione. Infatti qualora B fosse un predicato vero ma di fatto non dimostrabile dal sistema e Q un predicato banalmente falso, la prima regola negherebbe giustamente l'implicazione, mentre le ultime due l'affermerebbero.

Il linguaggio Loglisp, esteso con la possibilita` dei contesti, puo` essere utilizzato come strumento per la rappresentazione della conoscenza nei sistemi di Intelligenza Artificiale. Ovviamente si presta a rappresentare enunciati della logica del primo ordine, come pure conoscenza di tipo procedurale. Tuttavia il concetto di contesto bene si accoppia con l'uso dei frames, intesi come strumento per descrivere una conoscenza strutturata in moduli posti in

relazione tra loro. I contesti possono essere utilizzati per l'implementazione di un sistema a frames in due possibili modi.

In primo luogo se vogliamo seguire l'interpretazione di frame data da chi e` piu` vicino alle metodologie e al formalismo della logica, possiamo considerare l'approccio descritto in Nilsson [7] o, piu` ampiamente, in Hayes [4]. In quest'ottica i frames possono essere riformulati come collezioni di enunciati del primo ordine, provviste di un nome e contenenti relazioni con altri frames. E` chiaro come un contesto risponda perfettamente a questa definizione, non essendo altro che un insieme di enunciati raccolti sotto un certo nome. Inoltre la possibilita` di esprimere relazioni con altri contesti e` assicurata dalle primitive descritte sopra. Infatti mediante queste ultime e` possibile, associando ad ogni contesto una lista di contesti da cui e` permesso ereditare, fare in modo che qualora una deduzione fallisca sia ritentata in tutti i contesti specificati. In tal modo la risposta viene cercata dal particolare al piu` generale, con una struttura di contesti annidati: si tentera` di soddisfare una richiesta prima localmente nel contesto attivo, poi nei contesti dai quali questo puo` ereditare, quindi in quelli dai quali questi ultimi possono ereditare e cosi` via. Risulta cosi` definita una struttura, data al programma staticamente all'atto della sua creazione, che si oppone alla struttura dinamica dovuta all'uso delle primitive "stack" e "unstack", similmente a quanto avviene nei linguaggi di programmazione tipo-Algol. Il vantaggio di questo primo approccio all' interpretazione dei frames consiste nell' unire all' utilizzo di oggetti chiaramente strutturati e correlati tutto il potere deduttivo della logica. E` noto infatti che, sebbene siano un limpido strumento rappresentativo ed esplicativo, i frames risultano a volte non sufficientemente potenti dal punto di vista del ragionamento per inferenza.

Questo punto di vista rischia tuttavia di snaturare le idee da cui il concetto di "frame" e` partito, quando e` stato proposto da Minsky [3] come tentativo di teoria della conoscenza. D'altra parte e` utile distinguere (vedi Hayes [4]) l'interpretazione dei frames come linguaggio, da quella che li considera come stile di programmazione, concetto di fondo da porre alla base delle effettive implementazioni. Nel primo caso e` necessario elaborare un formalismo, ovvero identificare il concetto di frame con una sintassi e una semantica, cosi` come si fa per la logica del primo ordine o per il λ-calcolo. In quest'ottica ci si accorge che i frames non aggiungono espressivita` o potere deduttivo alla logica. Infatti ogni frame puo` essere riformulato senza perdita di informazione come insieme di enunciati raccolti con l'uso di un predicato aggiuntivo che ne esprime la correlazione. Nel secondo caso invece, non sarebbe corretto effettuare una simile riduzione, poiche` "nell'intenzione" il concetto di frame si differenzia profondamente dall'approccio della logica. Inoltre in molti sistemi sarebbe stato poco opportuno richiedere all'utente di tradurre in logica una conoscenza spesso naturalmente descrivibile in frames. In questi casi risulta vantaggioso mantenere una struttura a frames e ovviare allo scarso potere deduttivo aggiungendo alcuni "slots" che contengono come valore regole di produzione. Le modalita` e il momento in cui tali regole verranno applicate possono essere specificate da un controllo esterno. E` questo l'approccio seguito, ad esempio, nel sistema Centaur [8]. Cosi` la struttura globale resta quella di un sistema a frames, dove la conoscenza che e` necessario esprimere in logica rimane isolata in punti determinati. Nel caso del Loglisp e` possibile combinare l'uso dei contesti con un linguaggio a frames, permettendo di specificare come valore di uno slot il nome di un contesto.

Seguendo questa scelta la versione di Loglisp descritta in questo lavoro e` stata integrata con un sistema a frames e applicata in [6]. I frames sono stati

qui implementati come descritto in Winston [5], aggiungendo la possibilita` di associare ad ogni frame uno o piu` contesti, che possono essere creati anche dinamicamente. Il controllo ad alto livello e` gestito da alcune funzioni Lisp, mentre i frames hanno essenzialmente la funzione di organizzare e strutturare i dati. La base di conoscenza e` scritta in Loglisp e suddivisa nei contesti associati ai frames.

4. UN ESEMPIO DI PROGRAMMAZIONE IN LOGLISP

Combinando Lisp e Logica e utilizzando in modo essenziale i contesti e` possibile rappresentare in modo del tutto naturale conoscenze che sarebbe difficile esprimere altrimenti. Specialmente quando, di una certa situazione, sono presenti punti di vista diversi e interagenti, l'utilizzo di basi di conoscenza separate permette di costruire un sistema flessibile e potente.

Si abbiano ad esempio i due seguenti enunciati:

"Mario pensa che l'auto valga 7 milioni"

"Giovanni sa che ne vale 5 ma gliela vende per 7".

Si voglia rappresentare la situazione qui descritta, con l'aggiunta di alcune conoscenze generali che permettano al sistema di effettuare semplici ragionamenti. Una possibile soluzione e` data dai tre contesti descritti in Fig. 4, relativi ai punti di vista di Mario, di Giovanni e del narratore.

```
Base

(vende Giovanni Mario auto 7)
(truffa x y) <- (vende x y w z)
                (pensa y (vale w z))
                (sa x (vale w t))
                (greaterp z t) -
(pensa x y) <- (equal x activectx) y -
(sa x y) <- y (pensa x y) -
```

```
Mario

Inherit-from: Base

(vale auto 7)
```

```
Giovanni

Inherit-from: Base

(vale auto 5)
(pensa Mario x) <- (stack)
                   (use Mario)
                   x
                   (unstack) -
```



```
Narratore

Inherit-from: Base

(vale auto 5)
(pensa x y) <- (stack)
               (use x)
               y
               (unstack) -
```

Il contesto "Base", comune ai tre, contiene il fatto che Giovanni vende l'auto a Mario per 7 milioni, la definizione di "truffare" e "sapere", e il fatto che ognuno conosce cio` che pensa. Nel contesto del narratore la regole che ha come conseguente (pensa x y) definisce in modo naturale come si puo` mostrare che x pensa y: - salva il contesto attuale
- usa il contesto di x (guarda le cose dal punto di vista di x)
- dimostra y
- ripristina il contesto salvato

Si capisce ora come in Fig. 4 sia rappresentato il fatto che il narratore conosce cio` che pensano i personaggi della storia, e Giovanni sa cio` che pensa Mario ma non viceversa. Il sistema e` in grado di dedurre ad esempio che

(truffa Giovanni Mario) e` vero,
(sa Giovanni (truffa Giovanni Mario)) e` vero,
(sa Mario (truffa Giovanni Mario)) e` falso.

Inoltre puo` distinguere concetti come truffare, pensare di truffare, sapere di truffare. Questo esempio, pur avendo carattere puramente dimostrativo, mette in rilievo la duttilita` e la naturalezza d'impiego di questo linguaggio.

Referenze

[1] R.A.Kowalski: "Logic for problem solving", Elsevier North Holland, New York, 1979.

[2] J.A.Robinson, E.E.Sibert: "LOGLISP: an alternative to PROLOG", in Machine Intellicence 10, 399-419, 1982.

[3] M.Minsky: "A Framework for Representing Knowledge", in The Psycology of Computer Vision, P.H.Winston (edt), McGraw-Hill Book Co., N.Y. 1975.

[4] P.J.Hayes: "The Logic of Frames" in The Frame Reader, De Gruyter, Berlin, 1979.

[5] P.H.Winston, B.K.P.Horn: "Implementing Frames", in Lisp, Addison-Wesley Pub. Co., N.Y. 1984.

[6] F.Bergadano, A.Giordana, P.Laface, L.Saitta: "A Knowledge-based Approach to Pattern Interpretation: Knowledge Use and Acquisition", Proc. 2nd Int. Conference on Advances in Pattern Recognition and Digital Techniques, Calcutta 1986.

[7] N.J.Nilsson: "Principles of Artificial Intelligence", Springer-Verley, 1980.

[8] J.S.Aikins: "Prototypical knowledge for Expert Systems", Artificial Intelligence 20, 163-210, 1983.

PROGRAMMAZIONE AD OGGETTI IN PROLOG

Paola Mello
Antonio Natali

D.E.I.S.
Universita' di Bologna

ABSTRACT

Il principale obiettivo di questo lavoro e' quello di introdurre un insieme di concetti e meccanismi per permettere la definizione di un programma come un insieme di 'oggetti' Prolog interagenti. La specifica delle interconnessioni statiche o dinamiche fra tali oggetti viene espressa ad un diverso livello di programmazione (meta-programma). I concetti e i meccanismi proposti sono considerati un punto di partenza per la introduzione di uno stile di programmazione ad oggetti in Prolog, nella convinzione che tali stile agevoli l' uso di tale linguaggio per applicazioni complesse e di sistema.

INTRODUZIONE

L' uso sempre crescente degli elaboratori per la soluzione di problemi non numerici e il grande interesse industriale allo sviluppo di Sistemi Esperti [1], accresce l' interesse su modelli computazionali di tipo dichiarativo, fondati sulla logica.

Il linguaggio Prolog [2] in particolare, grazie alla sua potenza espressiva e alla ormai dimostrata possibilita' di efficiente implementazione, si propone oggi come uno strumento promettente in questo settore per lo sviluppo di un ampio spettro di applicazioni. Tuttavia il Prolog non puo' essere oggi considerato un serio antagonista, almeno in ottica industriale, a linguaggi piu' tradizionali. Esso e' ancora privo infatti di un insieme di concetti e strumenti ormai consolidati in altri linguaggi per garantire una piu' rapida produzione ed una piu' facile manutenzione del prodotto software. Risultano in particolare assenti costrutti e meccanismi che consentano di esprimere concetti quali modularita', protezione e strutturazione della base di conoscenza in accordo ad astrazioni caratteristiche del dominio applicativo.

Questo lavoro e' stato svolto nell'ambito di una convenzione fra ENIDATA S.p.A e D.E.I.S. Bologna .

Tali concetti giocano un ruolo essenziale sia nella programmazione di 'sistema', che nel progetto di applicazioni complesse e hanno trovato una interessante sintesi nel modello ad oggetti, a cui stanno convergendo, anche se con motivazioni e interpretazioni differenti, molte aree di ricerca tradizionalmente separate fra loro, quali quelle dei Linguaggi di Programmazione, dei Sistemi operativi e dell'Intelligenza Artificiale. Nel campo dei Linguaggi di Programmazione, ad esempio, il concetto di tipo di dato astratto conduce alla visione di un programma come collezione di oggetti [3], istanze di precise astrazioni. Per i progettisti di Sistemi Operativi, un sistema e' spesso concepito come una collezione di risorse o di gestori di risorse in competizione o in cooperazione tra loro [4]. Nel campo dell'Intelligenza Artificiale notevole importanza hanno i modelli di sistema basati su conoscenze organizzate in unita' quali i frames [5].
E' nostra convinzione, dunque, che il modello ad oggetti possa costituire un valido paradigma di riferimento per la definizione di estensioni al Prolog capaci di migliorarne le capacita' espressive sia per applicazioni di sistema che per programmazione applicativa 'in-the-large', senza rinunciare agli innegabili vantaggi dello stile di programmazione dichiarativo.

L' obiettivo che qui ci si propone e' quello di spezzare la attuale monoliticita' del Prolog, consentendo la definizione di programmi come collezioni di unita' (dette P-units) interagenti e separate tra loro non solo concettualmente ma anche a tempo di esecuzione. Questa estensione introduce un 'terzo livello' di modularita', che si aggiunge ai due gia' esistenti nel Prolog, rappresentati dalla separazione tra base di conoscenza e controllo e dalla autonomia che ogni fatto o regola ha rispetto agli altri. Questo terzo livello intende essere lo strumento primario per la definizione di programmi o sistemi software complessi. Un sistema puo' essere definito infatti come una collezione di unita' di conoscenza, ciascuna specializzata in un certo dominio applicativo e capace di cooperare con altre, delegando loro la dimostrazione dei predicati di relativa competenza. Ogni P-unit e' un mondo 'aperto' [6], non necessariamente statico nel tempo e capace di interazioni con altri per domandare la dimostrazione di particolari 'goals'; allo stesso tempo pero' ogni unita' puo' essere capace di nascondere la collezione delle proprie conoscenze rendendo visibili solo alcuni 'predicati di interfaccia'. Sistemi organizzati in tal modo possono avvalersi di supporti a tempo di esecuzione di tipo distribuito e aprono la possibilita' a riconfigurazioni dinamiche.

Lo scopo non e' quello di definire un nuovo linguaggio, ma di esprimere i precedenti concetti mantenendo col linguaggio Prolog piena compatibilita', nella convinzione che esso sia attualmente il riferimento obbligato nel settore.

1. CONCETTI E MECCANISMI DI BASE

1.1 OGGETTI E META-OGGETTI

Una P-unit e' formata da:
1) un insieme di clausole Prolog che rappresentano una base di conoscenza oggetto (OKB) su un particolare dominio;

2) un a Macchina Base (BM) o interprete in grado di rispondere a richieste di dimostrazione di 'goals' o 'sub-goals'.
Ogni tentativo di dimostrazione di un goal in una particolare P-unit U crea una nuova istanza di U il cui nome puo' essere definito dall'utente o dal sistema.
Il punto critico a questo proposito consiste nella definizione dei meccanismi di comunicazione tra P-units. Nella presente proposta tali meccanismi si fondano su una organizzazione di base capace di associare ad ogni P-unit una meta-P-unit, in cui il programmatore puo' incapsulare specifiche sulle modalita' di dimostrazione dei goals di livello oggetto; in particolare una meta-P-unit ha la capacita' di 'dirigere' la dimostrazione di un goal a precise P-unit, in modo da realizzare politiche e protocolli di comunicazione.

Ogni volta che la macchina base di una P-unit deve dimostrare un goal o sottogoal g1, richiede la dimostrazione del seguente meta-goal alla meta-P-unit associata:

todemo(Sender,Goal,Res)

dove Goal e' istanziata ad una rappresentazione interna (ad esempio una lista) di g1, Sender rappresenta l'istanza mittente e Res il risultato della dimostrazione del goal.
I risultati della dimostrazione del meta-goal todemo possono essere i seguenti:
a1) todemo e' dimostrato e Res e' istanziata a true; anche g1, di cui alcune variabili possono essere state legate durante il processo di dimostrazione, e' considerato dimostrato dalla macchina base;
a2) todemo e' dimostrato, ma Res e' istanziato ad un valore diverso da true (ad esempio false, unknown etc.); g1 e' considerato non-dimostrato e tale risultato e' comunicato al chiamante (se g1 era una richiesta esterna) o induce backtracking nella OKB (se g1 era un sotto-goal);
a3) todemo fallisce; la meta-P-unit non sa come dimostrare g1; g1 e' considerato non-dimostrato anche a livello oggetto (analogo al punto a2)) .

Per decidere come dimostrare un goal di livello oggetto la meta-P-unit puo' adottare una delle seguenti politiche:
1) risolvere il goal direttamente;
   ad esempio per la risoluzione del goal isless (e' minore di) definendo:
   todemo(Any,[isless,1,2],true).
   oppure:
   todemo(Any,[isless,2,1],false).
2) richiedere la risoluzione del goal ad un'altra P-unit utilizzando un predicato predefinito:
   send(Dest,Goal,Res)
   che e' dimostrato con successo quando la macchina base relativa alla P-unit identificata dal parametro 'Dest' ha terminato il tentativo di dimostrazione del goal rappresentato dal parametro 'Goal'; Res e' istanziato al risultato di tale dimostrazione (true, false, unknown etc.). A livello implementativo tale primitiva puo' essere interpretata come uno scambio di messaggi sincrono fra l'attivazione della P-unit chiamante e la P-unit destinataria.

   Ad esempio mediante:
   todemo(Any,[isless,X,Y],Res):- send(integer,[isless,X,Y],Res).
   la richiesta di dimostrazione del goal 'isless' e' inviata alla P-unit 'integer'.
3) utilizzare la propria P-unit oggetto per la dimostrazione del goal. Tale politica e' espressa mediante il predicato predefinito:
   demo(Goal,Res).
   Ad esempio mediante:
   todemo(Any,[isless,X,Y],Res):- demo([isless,X,Y],Res)
   la dimostrazione del goal 'isless' e' eseguita utilizzando la OKB della P-unit oggetto.

Poiche' anche una meta-P-unit e' una P-unit, anche ad essa puo' essere associata una meta-P-unit etc.. I meta-livelli possono dunque aumentare indefinitivamente. La suddivisione di base tra diversi livelli computazionali o descrittivi arricchisce notevolmente le capacita' espressive di un linguaggio, come ormai ampiamente dimostrato [7], [8]. Come esemplificato piu' estesamente in [9] la definizione di questi semplici meccanismi base consente di esprimere elegantemente linking statico o dinamico fra P-units, forme di "query the user", debugging ad alto livello di programmi, leggi di ereditarieta' fra P-units etc. programmando opportunamente le rispettive meta-P-units.
Dal punto di vista dalla organizzazione del software, l'organizzazione proposta consente anche una notevole riusabilita' [10] delle P-units. Sistemi molto diversi tra loro possono essere ottenuti partendo dalle stesse unita' e modificando i relativi metalivelli.
Ovviamente il problema che occorre affrontare e' in questo caso quello della efficienza e dell' overhead a tempo di esecuzione. Questo problema e' strettamente correlato all' architettura del supporto a tempo di esecuzione e si presenta qualitativemente diverso in sistemi monoprocessore o in sistemi distribuiti. Implementazioni in ambiente monoprocessore sono in corso di simulazione e realizzazione su personal computers e macchine Sun.

1.2 PROCESSI E LORO INTERAZIONE

Unita' separate di conoscenza possono essere concepite come unita' passive, capaci di rispondere concettualmente ad ogni richiesta esterna in qualunque momento, o come unita' attive, dotate concettualmente di un proprio flusso computazionale indipendente da quello delle altre e capaci di decidere esplicitamente la modalita' di risposta alle richieste esterne, sulla base dei valori assunti da certe variabili logiche locali. P-units attive possono essere viste come 'processi' di sistemi piu' tradizionali, dotate di una 'memoria' interna (rappresentata dal valore delle loro variabili logiche) e quindi capaci di modellare, in un ambiente puramente logico e privo di effetti collaterali, un concetto di stato.

Nella presente proposta un processo puo' essere creato dinamicamente mediante l'invocazione del predicato di sistema:

start(U, P, IG)

che crea una nuova istanza (processo) per la P-unit U con nome [U,P] e goal iniziale IG. L'istanza che ha invocato il goal start

non deve attendere la dimostrazione del goal IG, ma prosegue subito dopo l'inizio della sua dimostrazione.

La comunicazione e sincronizzazione fra processi e' ottenuta attraverso opportune 'clausole di sincronizzazione' ispirate da un lato alla Distributed Logic [11] e dall' altro al modello a rendez-vous [12] che hanno la seguente forma:

entry(....),accept(.....) :- body(...).

'entry' rappresenta un predicato di interfaccia invocabile da altre P-unit e 'accept' un predicato interno al processo, non visibile all' esterno. L'unificazione della testa della clausola di sincronizzazione rappresenta la confluenza di due flussi di controllo: quello del processo chiamante e quello del processo chiamato. Essa avviene con successo se e solo se entrambi i goals 'entry' e 'accept' sono stati invocati e unificano con successo. Se tale unificazione fallisce o se fallisce il 'body', viene innescato il backtracking in entrambi i processi comunicanti. Altrimenti sia 'entry' che 'accept' sono considerati dimostrati con successo. Il fallimento del predicato di interfaccia implica in generale il backtarcking del chiamante in quanto il processo chiamato non e' in grado in quel momento di dimostrare con successo quel goal. Tale dimostrazione potrebbe pero' avvenire nel futuro, visto che il processo chiamato e' dotato di uno stato che evolve. Il chiamante potrebbe percio' non semplicemente chiedere la dimostrazione del predicato entry, ma pretenderla, attendendo di conseguenza il successo di tale dimostrazione. Tale politica (caratteristica delle comunicazioni tra processi in linguaggi tradizionali) puo' essere realizzata ritrasmettendo (o riaccodando) la richiesta del chiamante quando si verifica il fallimento del predicato 'entry'. Se, per il processo che invoca il goal 'accept' non ci sono richieste esterne unificabili, esso puo' essere temporaneamente sospeso in loro attesa.

La OKB di un processo buffer potrebbe ad esempio essere definita nel seguente modo:

```
(1) buffer(B):- notempty(B),remove(B,C)
                buffer(C).
(2) buffer(B):- notfull(B), insert(B,C),
                buffer(C).
(3) buffer(B):- buffer(B).
(4) get(X), remove(B,C):-  out( X,B,C ).
(5) put(X), insert(B,C):-  in( X,B,C ).
(6) init(X) :- buffer(X).
```

dove 'put' e 'get' costituiscono i predicati interfaccia. Internamente il buffer e' un loop senza fine che presenta due punti di sincronozzazione con l'esterno ('remove' e 'insert'). I Predicati 'notempty' e 'notfull' possono essere interpretati come guardie locali che controllano il non-determinismo delle comunicazioni. La clausola (3) viene selezionata quando falliscono le prime due, cioe' quando nessuna richiesta esterna puo' essere servita. Lo stato del buffer e' rappresentato dalle variabili locali B e C.

Produttori e consumatori potrebbero essere introdotti come segue:

```
unit( produttore ).
        prod:- read(CH), put(CH), prod.
unit( consumatore ).
        cons:- get(CH), write(CH), cons.
```

2. VERSO UNA PROGRAMMAZIONE AD OGGETTI

Secondo una interpretazione ad oggetti le P-units potrebbero essere considerate come tipi o classi. Ogni predicato interfaccia costituisce un metodo. Nell' esempio precedente i predicati interfaccia 'put' e 'get' costituiscono i metodi definiti nella classe buffer. La definizione dei comportamenti espressa nei metodi puo' essere confermata, completata o modificata dalle meta-P-units associate.

Un particolare sistema puo' essere definito creando un insieme di oggetti istanze di particolari P-units. Se tali oggetti devono avere uno stato (per modellare ad esempio risorse fisiche), essi saranno definiti come processi. Il comportamento di un processo e' definito nella loro classe (P-unit), mentre lo stato caratteristico di ogni istanza (variabili istanza) e' racchiuso nelle variabili logiche di ogni singolo processo ed e' dunque diverso per ogni istanza.
Ad esempio, un particolare sistema costituito da quattro processi paralleli e perpetui chiamati rispettivamente prod1, buff1, buff2, cons1, potra' essere definito invocando i seguenti goals:

```
start( producer, prod1,  prod),
start( buffer,   buff2,  init( [] ) ),
start( buffer,   buff1,  init( [3] ) ),
start( consumer, cons1,  cons).
```

Le istanze buff1 e buff2 sono create con stati locali iniziali diversi. L'accesso disciplinato allo stato locale dei processi e' garantito dalla presenza delle clausole di sincronizzazione. In questo senso, a differenza di linguaggi ad oggetti quali [3], qui il concetto di processo e di accesso disciplinato alle risorse e' parte integrante del modello proposto. Le richieste di goals da dimostrare sono interpretabili come scambi di messaggi tra oggetti. A differenza di altri linguaggi ad oggetti, la politica di connessione fra le varie istanze puo' qui essere espressa a un livello diverso (in una meta-P-unit) da quello che esprime la specifica del comportamento (metodi) contenuta nella OKB degli oggetti, per garantire maggiore modularita' e flessibilita' nel sistema.
Una meta-P-unit per il buffer potrebbe essere la seguente:

```
(1') todemo(X,[out¦Z],Res)          :- !,send( list,[out¦Z],Res).
(2') todemo(X,[in¦Z],Res)           :- !,send( list,[in¦Z],Res).
(3') todemo(X,[notempty¦Z],Res)):- !,send(list,[notempty¦Z],Res).
(4') todemo(X,[notfull¦Z],Res)   :- !,send( list,[notfull¦Z],Res).
(5') todemo(X,Goal,Res)             :- demo(Goal,Res).
```

La P-unit 'list' viene invocata per risolvere tutti i goal relativi alla manipolazione della rappresentazione interna del buffer, men-

tre per gli altri goal viene utilizzata la OKB del buffer (5'). In questo modo la rappresentazione interna del buffer puo' essere facilmente cambiata.
I Predicati 'read'e 'put' dei produttori sono non locali e come dimostrarli e' specificato nella loro meta-P-unit:

```
todemo(X,[put¦Z],Res):- !,
            send( [buffer,buff1], [put¦Z], Res).
todemo(X,[read¦V],Res):- !,
            send( in-out, [read¦V], Res).
todemo(X,Goal,Res):- demo(Goal,Res).
```

In questo caso ogni istanza del produttore e' collegata a priori con l'istanza del buffer 'buff1', ma e' permessa, con una opportuna programmazione delle meta-P-unit anche una connessione dinamica fra istanze ([3],[13]). Ad esempio:

```
todemo( X, [put¦Z],Res ):-
      is_integer(Z), send( [buffer,buff1],[put¦Z],Res ).
todemo( X, [put¦Z], Res ):-
      send( [buffer,buff2],[put¦Z],Res ).
```

specifica una connessione dei produttori con buff1 per dati di tipo intero e con buff2 per dati di altro tipo.
Tale specifica potrebbe essere automaticamente creata da un opportuna ambiente di programmazione sulla base di comandi di configurazione dettati dall' utente.

Normalmente, nei linguaggi ad oggetti, le classi possono essere organizzate secondo relazioni di ereditarieta: in Smalltalk-80 [3] e' possibile definire una forma di ereditarieta' semplice fra classi, di metodi e variabili; in LOOPS [13] tale forma di ereditarieta' puo' essere multipla.
Leggi e meccanismi di ereditarieta' sono sufficientemente consolidati tra oggetti passivi. Meno sistemato e' invece il concetto di ereditarieta' tra processi. Ad esempio, ereditare il comportamento da una super-classe processo significa solo ereditare la definizione dei predicati interfaccia ((4) e (5) nella OKB della P-unit buffer) o anche le guardie e la capacita' di pilotare la scelta delle alternative? ((1) (2) (3) nella OKB della P-unit buffer). Uno dei vantaggi della presente proposta e' di poter studiare il problema attraverso una opportuna programmazione delle meta-P-unit. Diversamente da altre proposte infatti, quali Mandala [14] o ESP [15], qui non si introduce un nuovo linguaggio dichiarativo basato sul modello ad oggetti ma piuttosto un meccanismo che utilizzando la potenza espressiva di un meta-livello consente la definizione di differenti ambienti a oggetti con differenti proprieta'.

CONCLUSIONI

La possibilita' di specificare un programma come una collezione di unita' Prolog separate e interagenti, dotate o meno di stato, definite dal sistema o dal programmatore, non solo permette modularita', protezione, parallelismo e capacita' di distribuzione, ma consente anche di esprimere programmi in accordo a quel modello ad oggetti che risulta oggi essere un denominatore comune di molti campi applicativi.

Nasce cosi' la possibilita' di definire mondi integrati, uniformi ed espandibili intorno a un nucleo centrale di oggetti tipico di un ambiente di programmazione anche nel contesto di sistemi che vogliano essere fondati su un paradigma di tipo dichiarativo. Una applicazione puo' essere cosi' immediatamente percepita come un sistema fondato sulla rappresentazione e manipolazione di conoscenze, ponendo l' enfasi su un criterio progettuale oggi ritenuto fondamentale non solo nel settore dei sistemi esperti.
Questo e' l' obiettivo di lungo termine, di sfondo alla presente proposta, di cui essa vuol essere solo un possibile punto di partenza.

RIFERIMENTI

[1] R.Davis, D.Lenat :" Knowledge-Based Systems in Artificial Intelligence ", New York: McGraw-Hill, 1980.
[2] W.F. Clocksin, C.S. Mellish : " Programming in Prolog ", Springer-Verlag, New-York, 1981.
[3] A.Goldberg, D. Robson, "Smalltalk-80, The Language and its Implementation ", Addison Wesley, 1983.
[4] A.K.Jones:"The Object Model: A Conceptual Tool for Structuring Software", in 'Operating Systems' , ed. da Bayes et al., Springer Verlag n.60, 1978.
[5] M.Minsky: " A Framework for Representing Knowledge", in 'Psycology of Computer Vision', Winston ed. McGraw-Hill, 1975.
[6] C.Hewitt, P.De Jong :"Open Systems", Tech. Rep. MIT-AIM 691 Dicembre 1981.
[7] L.Aiello, G.Levi:" The uses of meta-knowledge in AI Systems", ECAI-84, Pisa, September 1984.
[8] K.Bowen, R.Kowalski :"Amalgamating language and metalanguage in logic programming", in Logic Programming , Academic Press, 1982.
[9] P.Mello,A.Natali:"Programs as Collections of Communicating Prolog Units", accettato al European Symposium on Programming, Saarbrucken, Marzo 1986.
[10] P. Wegner :"Capital Intensive Software Technology" , IEEE Software , v.1, n.3, Luglio 1984.
[11] L.Monteiro :" A Proposal for Distributed Programming in Logic", Tec. Rep. University of Lisbona, Gennaio 1983.
[12] "Reference manual for the Ada programming language ", U.S. Departement of Defense, ANSI/MIL-std 1815-a, Gen.1983.
[13] D.G. Bobrow, M.Stefik : "The LOOPS Manual ", Xerox Corporation, Gennaio 1983.
[14] K.Furukawa et alii: "Mandala: A Logic Based Knowledge Programming System", in International Conference On Fifth Generation Computer Systems 1984.
[15] T.Chikayama: "ESP Reference Manual", ICOT Report, Febbraio 1984.
[16] "Technical Annex To Esprit Project No 363: Advanced Logical Programming Environments", 1985.

L'USO DI STRATEGIE FLESSIBILI COME MECCANISMO DI CONTROLLO DELL'INTERPRETAZIONE DEL PROLOG

Claudio Arbib - Gianna Cioni

Istituto di Analisi dei Sistemi ed Informatica
Viale Manzoni, 30 - 00185 Roma

SOMMARIO

Obiettivo di questo lavoro e' mostrare le caratteristiche principali che deve avere un linguaggio di programmazione per poter essere impiegato nella soluzione di problemi di decisione. Vengono considerati i linguaggi logici, ed in particolare il PROLOG, per derivare delle indicazioni per migliorare l'efficienza dei programmi, mantenendo costante la correttezza e la leggibilita'. La linea seguita e' quella di operare sul controllo tramite strategie flessibili e controllabili dall'utente stesso. In questo lavoro vengono presentati i primi risultati della ricerca in corso che si colloca anche nel progetto ESPRIT "Advanced Logical Programming Environments".

INTRODUZIONE

Obiettivo del presente lavoro e' tentare di sintetizzare quali siano le principali caratteristiche necessarie a un linguaggio di programmazione logica per poter essere impiegato nella soluzione di quell'ampia classe di problemi, costituente uno schema generale per un gran numero problemi reali, che viene generalmente indicata tramite la locuzione "problemi di decisione".

Tali problemi sono quelli descrivibili per mezzo di un insieme di stati Q e di due suoi sottoinsiemi S e T la cui interpretazione e', rispettivamente, di insieme degli stati iniziali e insieme di quelli finali; a completare la descrizione del problema viene specificata una funzione di transizione che guidi l'evoluzione del sistema da stato a stato: tale funzione puo' non essere deterministica ed essere rappresentata sotto forma di condizioni che prescrivano il passaggio da un certo sottoinsieme di stati a un altro; pertanto, l'evoluzione del sistema puo' seguire in generale diverse sequenze di stati e, di fatto, il nome di "problema di decisione" e' dovuto proprio all'intrinseca necessita' di mettere a punto strategie (euristiche) che attuino opportuni criteri di scelta nei successivi stadi. Si perverra' a una soluzione per il problema allorche' si sara' determinato un percorso che, partendo da uno stato in S, giunga dopo un numero finito di passi a uno stato in T. Le problematiche relative all'approccio descritto riguardano l'esistenza di una soluzione, la sua unicita', il modo migliore, cioe' piu' efficiente, di determinarla.

Notiamo che problemi del genere che e' stato qui descritto sono di norma, per la loro stessa generalita', estremamente difficili da risolvere in modo puramente algoritmico: l'applicazione di euristiche di uso

generale puo' essere di grande aiuto, ma e' palese che tale applicazione va affiancata ad uno sfruttamento il piu' ampio possibile delle proprieta' del particolare problema, e che tale sfruttamento puo' avvenire solo a partire da una descrizione del modello estremamente accurata e attentamente interfacciata con l'euristica. Cio' giustifica l'interesse di recente sviluppatosi per la programmazione dichiarativa, in quanto essa, a differenza delle precedenti filosofie, evita una separazione netta fra modello e algoritmo svincolando lo specifico problema dalla strategia adottata per la sua risoluzione.

Infatti, la risoluzione di un generico problema, sia o meno esso puramente computazionale, con metodologie ordinarie richiede due fasi di elaborazione ben distinte: nella prima viene costruito un modello rispondente alle specifiche del problema in esame, mentre la seconda consiste essenzialmente nella messa a punto di un algoritmo in grado di risolvere il problema considerato.

La logica che caratterizza oggi i linguaggi di tipo imperativo, ed in particolare di quelli a molto alto livello, e' fondamentalmente quella dei tipi di dati astratti. In questo senso la prima fase della risoluzione di un problema concerne fondamentalmente l'organizzazione e la strutturazione dei tipi di dati coinvolti, mentre solo nella seconda fase, non sempre in modo intrinsecamente connesso ed integrato, viene affrontato il problema algoritmico.

L'uso di linguaggi dichiarativi, espressamente orientati alla descrizione del problema, dovrebbe invece consentire una caratterizzazione ben piu' completa del problema stesso, mettendo l'utente in condizioni di esprimerne la struttura per mezzo di relazioni fra gli oggetti che la compongono, relazioni che in generale devono poter essere interpretate dinamicamente. Proprio la presenza di una interpretazione dinamica delle relazioni descriventi la struttura del problema, fa si' che il paragone fra formalismi di diverso tipo conduca a una sostanziale diversita' di approccio fra i due casi per cio' che riguarda la fase algoritmica: infatti una descrizione del problema che faccia uso di un formalismo logico, e che sia corretta e completa in tale formalismo, gia' contiene in se' tutte le informazioni necessarie alla risoluzione del problema; pertanto, un interprete che sia in grado di processare le formule ben formate del formalismo, secondo le regole di inferenza e le proprieta' da esse espresse, realizza in pratica un algoritmo di risoluzione di uso generale.

Il seguente breve esempio puo' servire a illustrare il concetto. Poniamoci il problema di sommare due numeri interi: in un linguaggio imperativo, quale il Pascal, cio' corrisponde a definire una struttura di dati (che nel Pascal e' peraltro gia' definita) nella quale l'operazione somma abbia il senso che comunemente le viene dato nell'aritmetica; tale struttura di dati altro non e' che un modello della struttura algebrica costituita dall'insieme degli interi piu' lo zero, piu' l'operazione di somma, a sua volta definita tramite un certo insieme di assiomi. Tali assiomi risultano impliciti nella definizione operativa di somma, nel senso che la loro verifica e' subordinata alla correttezza dell'algoritmo che in Pascal realizza la somma di due numeri. Viceversa in PROLOG, come in altri formalismi logici, gli assiomi che definiscono la somma fra gli interi sono direttamente definibili tramite f.b.f. del formalismo: l'interpretazione di una qualsiasi f.b.f. del PROLOG che esprima la relazione intercorrente fra due addendi e la loro somma avviene allora senza bisogno di alcun algoritmo che realizzi effettivamente la somma in questione (di tale algoritmo, fra l'altro, ne dovrebbe esistere uno per

ogni schema di ingresso/uscita), bensi' per mezzo della sola applicazione delle regole di inferenza espresse dagli assiomi in questione.

Possiamo osservare che il PROLOG e' suscettibile di svariate 'letture'. Vi e' infatti una sua interpretazione procedurale che deriva dal considerare le clausole come procedure che si richiamano reciprocamente passandosi tra di loro parametri i cui schemi di ingresso/uscita vengono, nel PROLOG ordinario, fissati al momento dell'esecuzione Ma vi e' anche un'interpretazione che fa delle clausole altrettanti processi paralleli che stabiliscono le reciproche comunicazioni attivando canali temporanei tramite l'unificazione; in questo caso gli schemi di ingresso/uscita vengono fissati staticamente ottenendo cosi' un'implementazione che non fa uso del backtracking. Il PROLOG automatizza, nella sua forma piu' semplice, quella parte del Calcolo Predicativo del Primo Ordine che va sotto il nome di Clausole di Horn: formalmente, le Clausole di Horn discendono dalla forma clausale del Calcolo del Primo Ordine con l'ipotesi aggiuntiva che in ciascuna di esse non sia presente piu' di un letterale positivo; rispetto ad esse, il procedimento di risoluzione adottato (ordered input resolution) si dimostra essere corretto e completo. Tale procedimento e' rigidamente messo in atto dall'interprete del linguaggio, senza possibilita' di deroghe relativamente alla scelta delle clausole da processare. In realta' alcune primitive del PROLOG ordinario consentono di estendere l'espressivita' del formalismo, ad esempio rendendo possibile la negazione di clausole "per fallimento". Altre primitive, dette di "meta-livello", (call, atom, var, clause, assert ecc.) rendono possibile la realizzazione di meta-interpreti che trattano clausole come termini all'interno di altre clausole: pertanto e' possibile, ad esempio, definire un meccanismo di interpretazione delle clausole che operi a livello superiore in modo piu' flessibile dell'interprete di base. Tuttavia cio' viene fatto, come del resto appare ovvio, a prezzo di notevoli inefficienze. Vi sono infine primitive grazie alle quali viene data ad alcuni termini un'interpretazione algebrica (is, eval ecc.), in modo da poter trattare agevolmente l'aritmetica senza dover far ricorso a una definizione assiomatica.
Se e' fuori di dubbio l'utilita' di tali primitive nei contesti applicativi, molto meno accettata e' la loro confusa intrusione nel substrato logico del formalismo da chi ritiene il conservare la purezza di tale substrato come la migliore garanzia per la semplicita' di sviluppo del modello a partire dalle specifiche e la sua correttezza nei riguardi di esse.

Dal punto di vista metodologico si e' partiti quindi da una critica al PROLOG, visto, nonostante tutte le proposte di modifiche piu' o meno sostanziali, come il piu' affermato fra i linguaggi di programmazione logica, per giungere a individuarne le carenze sotto il profilo della flessibilita' di programmazione che lo rendono a tutt'oggi inadeguato a trattare la maggior parte dei problemi di interesse pratico. Tale critica e' stata peraltro non a caso condotta facendo riferimento a "toy problems", per mostrare che le inadeguatezze riscontrate non dipendono tanto dalla complessita' del problema in esame, quanto dalla rigidita' dei meccanismi di interpretazione e dalla volonta' di mantenere il formalismo stesso a un livello piu' basso e generale possibile. Questa volonta' si mostra d'altro canto in contrasto sia con le aspirazioni di chi voglia conferire alla programmazione dichiarativa una diffusione piu' ampia di quella attuale, limitata per lo piu' a istituti di ricerca e ad applicazioni ad alto contenuto tecnologico, sia con le istanze di miglior trade-off fra correttezza e leggibilita' dei programmi da un lato e loro efficienza dall'altro.

STILE DI PROGRAMMAZIONE

In generale, sarebbe logico attendersi in un linguaggio dichiarativo la stessa possibilita' di sviluppare programmi corretti e leggibili adottando tecniche di programmazione strutturata top-down ovvero bottom-up, analogamente a quanto e' reso possibile in un certo insieme di linguaggi di tipo imperativo. Cio' equivale in sostanza, nel caso del PROLOG, a richiedere una certa separazione fra le specifiche piu' generali del problema in esame, le regole e proprieta' relative agli oggetti costituenti la struttura del problema, e le vere e proprie strutture di dati utilizzate (i tipi di dati di base del PROLOG, cioe' i termini, trattati eventualmente come liste, piu' le operazioni su di essi definite). A grandi linee, il primo dei tre livelli descritti andrebbe considerato come quello "dichiarativo" per antonomasia, mentre l'ultimo dovrebbe corrispondere a un "hardware" implementato, in PROLOG, dall'algoritmo di unificazione.

In accordo con la ben nota definizione, dovuta a Kowalski, che descrive le componenti principali di un programma, avremo a che fare con:

- Strutture di dati;
- Logica del problema;
- Controllo della esecuzione.

Essendoci limitati a considerare problemi di decisione, coerentemente alla definizione di questi data nell'Introduzione, potremo ragionevolmente prendere in esame, rispettivamente:

- Mosse da uno stato a un altro;
- Regole che definiscono quali mosse siano da intendersi corrette;
- Trucchi, Tattiche e Strategie per trovare una sequenza di mosse corrette che conduca ad una soluzione.

Piu' in dettaglio, una mossa valida sara' costituita da una coppia di stati ricavati in base ad un insieme di azioni eseguite correttamente secondo un certo insieme di regole. Ognuna di queste azioni avra' a sua volta come dominio un certo insieme di dati elementari rispetto agli stati (i quali sono, al pari dei primi, termini del PROLOG) e, in PROLOG, verra' materialmente eseguita per mezzo dell'algoritmo di unificazione, che agisce appunto su termini; in questo senso, dati e azioni formano insieme una sorta di "struttura di dati". D'altro canto, trucchi, suggerimenti (dati dall'utente al sistema) e tattiche di vario genere dovrebbero ritenersi conoscenze aventi carattere differente da quello delle regole del problema, le quali ultime ne esprimono puramente e semplicemente la struttura attraverso relazioni fra gli "oggetti" che la compongono. Pertanto, almeno dal punto di vista dello stile di programmazione, si puo' agevolmente sostenere che piu' netta e' la separazione fra i due tipi di conoscenza, migliore, piu' generale e piu' aderente alle specifiche ne risulta la descrizione del problema.

Per illustrare quanto detto, esaminiamo il seguente semplice programma PROLOG che affronta la risoluzione del problema della Torre d'Hanoi in modo quanto piu' possibile dichiarativo:

```
path(X,X).

path([A,B,C],[R,S,T]) :- mossa([A,B],[D,E]),
                         path([D,E,C],[R,S,T]);
                         mossa([A,C],[D,E]),
                         path([D,B,E],[R,S,T]),
                         ......

mossa([A,B],[C,D])    :- nil(B), car(A,Al),
                         rem(A,C), cons(Al,B,D).

......

nil(X)                :- X=[].

......
```

L'esempio riportato dovrebbe rendere chiaro cio' che intendiamo per mosse, azioni, regole e dati: le azioni (nil, car ecc.) sono di fatto eseguite dal motore inferenziale in accordo con le regole d'inferenza, in maniera tale che i dati rappresentanti gli stati della computazione, nel nostro semplice caso si tratta di liste, vengano istanziati tramite unificazione ai valori corretti e conseguentemente una mossa valida, se esiste, venga portata a compimento. Una catena di mosse valide costituisce un percorso (path): tale percorso viene generato nel nostro esempio per ricursione, al livello piu' alto dal punto di vista del dettaglio delle specifiche. Cio' che si vuole far qui notare e' l'aderenza dello schema proposto per l'uso del PROLOG ai problemi della Teoria dei Giochi e, piu' in generale, al mondo dei problemi di decisione sopra descritti.

STRATEGIE FLESSIBILI

Tornando per un attimo all'esempio proposto nel precedente paragrafo, e' facile verificare che, sebbene le regole che stabiliscono la correttezza delle mosse siano ben definite, il meccanismo di interpretazione puo' non condurre alla determinazione di un percorso finito: e, d'altro canto, nessuna regola e' stata imposta allo scopo di evitare una simile evenienza. Tuttavia dovrebbe essere chiaro che il problema del riconoscimento della terminazione andrebbe non affrontato a livello di "regole del gioco", ma piuttosto considerato come una questione inerente la strategia di ricerca di un percorso-soluzione. Qui di seguito riportiamo alcuni approcci alternativi al problema, sempre riferendoci all'esempio della Torre d'Hanoi:

- Ridefinizione del goal: secondo questo approccio, una soluzione corretta viene ridefinita come un percorso che termini:

```
solution(Probl,Path) :- path(Probl,Path), no_loop(Path).
```

Le strutture di dati usate in questo approccio differiscono da quelle della precedente formulazione: qui viene infatti introdotto un oggetto "Path" che altro non e' se non una lista di mosse eseguite correttamente. Se pure la leggibilita' del programma non ne risulta in effetti sacrificata in alcun modo, questa soluzione non puo' essere applicata con successo servendosi del PROLOG classico, dal momento che essa richiede una processazione parallela delle clausole (coroutining) che preveda fra di esse specifiche relazioni di tipo produttore-consumatore; difatti, ogniqualvolta una nuova istanza di

Path viene prodotta dal processo-clausola "path", l'attivazione di "no-loop" provvede a consumarla allo scopo di segnalare la presenza di eventuali cicli e far conseguentemente fallire il goal.

- Spostamento a un livello piu' basso del riconoscimento di eventuali cicli: la clausola "no-loop" viene in questo caso inserita direttamente fra le condizioni che caratterizzano il goal del programma primitivo:

```
path([A,B,C],[R,S,T],Done) :- mossa([A,B],[D,E]),
                              no_loop([D,E,C],Done),
                              cons([D,E,C],Done,New),
                              path([D,E,C],[R,S,T],New).

......

no_loop(T,[]).

no_loop(T,[T|_])          :- !,fail.

no_loop(X,[_|C])          :- no_loop(X,C).
```

E' immediato rilevare che in questo caso il controllo viene eseguito tramite la definizione di una regola, "no-loop" appunto, la quale opera su oggetti che non sono in realta' direttamente coinvolti nel gioco, e cioe' le due liste Done e New che contengono l'elenco delle mosse gia' fatte. E' come se dessimo al sistema il seguente suggerimento: per scegliere fra le varie possibili mosse corrette, prendi un taccuino e annota tutte quelle gia' fatte badando bene di non ripeterne nessuna. In effetti da un punto di vista strettamente dichiarativo, che si limiti cioe' a prendere in considerazione le sole specifiche del problema, un tale artificio non e' un gran che pertinente; a cio' si aggiunga che, sotto l'aspetto della strutturazione del programma, quale e' stata descritta nel paragrafo precedente, l'approccio in esame ci costringe a utilizzare un'"azione" (cons) allo stesso livello delle "regole del gioco" che definiscono il percorso.

- Un metodo alternativo, e forse in un certo senso piu' "pulito" del precedente, potrebbe essere il seguente: non appena una nuova mossa viene trovata, e posto che essa non sia gia' stata eseguita in precedenza, la si esegue e si aggiorna la base delle conoscenze aggiungendovi il fatto "just-done" come se fosse un teorema provato; dopo cio', si prosegue la ricerca di una nuova mossa e cosi' via sino ad aver completato il percorso:

```
path([A,B,C],[R,S,T]) :- mossa([A,B],[D,E]),
                         not( just_done([D,E,C]) ),
                         assert( Just_done([D,E,C]) ),
                         path([D,E,C],[R,S,T]).
......
```

Con questa soluzione evitiamo la definizione sia di strutture di dati addizionali e non inerenti la natura del problema sia di procedure che su di queste agiscano (come la "cons" introdotta nel caso precedente). Questa tecnica puo' essere ritenuta discutibile quanto a "pulizia" ed efficienza (quest'ultimo aspetto e' tuttavia legato al modo in cui l'algoritmo di risoluzione tratta la ricerca delle clausole nella base della conoscenza), ma presenta il vantaggio di far riflettere sul come le conoscenze relative alle strategie di

risoluzione vadano riferite a un contesto che si trova logicamente in posizione di meta-livello rispetto a quello delle specifiche del problema.
Altro discorso puo' senz'altro riguardare la "pericolosita'" intrinseca dell'effettiva messa in opera di tecniche simili in applicazioni a modelli di problemi reali: particolari accorgimenti, quali la ritrattazione delle asserzioni eseguite, vanno presi riguardo al backtracking, e in ogni caso all'insieme delle tecniche da utilizzare e' senz'altro richiesto un grado di sofisticazione ben maggiore di quello esemplificato nel caso presente.

Il problema della non terminazione di programmi PROLOG (ovviamente, si fa qui riferimento a casi particolari di tale problema!) e', fra quelli inerenti le strategie flessibili per l'interpretazione delle clausole, uno dei piu' studiati (si vedano ad es. [2] e [6]). Nonostante cio', la messa a punto di una strategia di uso generale che sia efficiente o che preservi la completezza dello spazio delle soluzioni ammissibili non sembra facilmente ipotizzabile [7]. D'altro canto il voler affrontare problemi di decisione servendosi di un linguaggio come il PROLOG, a causa della stessa struttura presentata da tali problemi, annovera fra i piu' frequenti e di piu' complessa risoluzione proprio il problema della non terminazione (mentre ad esempio un altro problema di strategia, quello della determinazione di tutte le possibili soluzioni e', in PROLOG, brillantemente superato grazie al backtracking) , e spesso un meta-interprete studiato ad hoc si rivela, nei casi di interesse pratico, come qualcosa di decisamente inefficiente. Tutto cio' va ad aggiungersi alle considerazioni riguardanti le difficolta' di realizzare meta-interpreti che siano davvero "puliti", mentre d'altra parte tecniche di meta-livello che modifichino la base delle conoscenze a tempo di esecuzione richiedono attualmente ancora un certo sforzo di approfondimento dei relativi aspetti teorici, correlate come sono a tematiche riguardanti non solo la robustezza ma anche la stessa correttezza dei programmi prodotti.

ULTERIORI PROBLEMI APERTI

Come osservato in precedenza, un programma PROLOG che sia "pulito" presenta varie distinzioni fra i diversi livelli di specificazione; grosso modo, tali livelli potrebbero venire descritti come segue:

- Specifiche generali (ricursione, relazioni fra oggetti)
- Specifiche intermedie (relazioni fra oggetti, procedure)
- Specifiche di basso livello (termini PROLOG, unificazione)

Fra parentesi sono state indicate alcune parole chiave che caratterizzano i vari livelli dal punto di vista della loro implementazione in PROLOG.

Uno schema siffatto non prevede l'uso delle cosiddette "caratteristiche spurie" (impure features) del linguaggio PROLOG, con particolare riferimento a quei predicati di sistema che assicurano un'efficiente valutazione di funzioni e/o espressioni aritmetiche (ad esempio il predicato "is"). Peraltro, l'uso di tali predicati diviene pressoche' inevitabile non appena il grado di complessita' del problema cresce ovvero la sua descrizione comporta l'interpretazione dei termini

base come elementi di una qualche struttura algebrica. A titolo di esempio, consideriamo il conosciuto rompicapo dei Cannibali e Missionari; un programma PROLOG che ne sia la rappresentazione potrebbe contenere la seguente regola che stabilisce le modalita' di spostamento senza danni dei protagonisti (si fa l'ipotesi che la canoa a disposizione non possa contenere piu' di 2 persone):

```
spostamento(N_pers,Stato_iniziale,Nuovo_stato) :-
                        /* POSTO CHE: */
                              tot_missionari(N),
                              Stato_iniziale = [M,C,Sponda],
                              Sponda = di_qua,
                              M_di_la is N-M,
                              C_di_la is N-C,
                        /* SE: */
                              N_pers =< C,
                              (M_di_la = 0;
                               M_di_la >= C_di_la + N_pers),
                        /* ALLORA: */
                              Nuovo_C is C-N_pers,
                              Nuova_sponda = di_la,
                              Nuovo_stato = [M,Nuovo_C,Nuova_sponda].
```

La particolare struttura della regola scritta mostra una distinzione fra le condizioni di applicabilita' della regola (che formano una specie di "guardia") e la regola di se' e per se' (che a sua volta e' posta in forma condizionale). Inoltre, dal momento che la natura del problema richiede l'uso dell'aritmetica, si e' resa necessaria l'applicazione del predicato "is".

In effetti, il linguaggio PROLOG non contiene "sovra-strutture" formali ne' del tipo di guardie ed espressioni condizionali (cio' che in realta', nei piu' diffusi formalismi PROLOG, e' vero solo in parte) ne' tali da consentire la trattazione di dati strutturati veri e propri (cioe' tali da rendere possibile l'interpretazione dei termini nell'ambito di strutture algebriche).
Questo secondo problema, fra i due di gran lunga il piu' importante, non puo' essere affrontato e risolto in modo completo senza una piu' estesa definizione del concetto di unificazione: tale sarebbe l'estensione dell'effetto del predicato PROLOG "=" aggiungendo ad esso la possibilita' di valutazione delle funzioni, con modalita' rigorosamente stabilite in modo assiomatico, alle ordinarie mansioni di matching bidirezionale con eventuale istanziazione di variabili.

In altri termini, nell'interpretazione usuale del PROLOG ogni termine rappresenta se stesso (minimal Herbrand model). La soluzione proposta (si veda ad es [4]) e' quella di far si' che i termini PROLOG vengano interpretati su di una Sigma-algebra definita da un insieme di equazioni (assiomi) che possono anche venire definite dall'utente. Tale scenario e' in effetti quello di una logica equazionale, nella quale i termini sono associati ciascuno al proprio tipo e dove gli assiomi vengono introdotti esattamente come clausole della base delle conoscenze, tipo

```
sum(X,0) = X.
```

Dal momento che, in generale, la soluzione generale di un sistema di equazioni non e' individuata da una singola sostituzione, esiste la possibilita' di determinare piu' di un "most general unifier" e cio' comporta particolari precauzioni nel controllo dei canali di comunicazione

fra clausole. Inoltre, mentre clausole PROLOG e clausole la cui testa non sia un'eguaglianza non creano problemi al meccanismo inferenziale, una generalizzazione che racchiuda anche clausole in cui un'eguaglianza dipenda da qualche altra clausola non e' facile da gestire, presentando il rischio di interferenze fra algoritmo di unificazione e resolution.

Un approccio del genere tuttavia comporterebbe i seguenti notevoli vantaggi:

1. non sarebbero necessari predicati spuri per la valutazione delle espressioni;

2. la tipizzazione forte introdurrebbe ulteriori garanzie di correttezza;

3. non vi sarebbe necessita' di alcun 'occur check' durante l'unificazione (nel caso ordinario, tale controllo e' necessario al fine di evitare computazioni infinite in sostituzioni del tipo X=f(X)).

Come si vede, dunque, i problemi aperti posti dallo studio delle possibilita' di applicazione di tecniche di programmazione logica a problemi di decisione non si limitano a questioni di carattere qualitativo nella stesura dei programmi: anche questioni sostanziali implicanti modifiche a livello di formalismo vengono sollevate, rispondenti a esigenze non solo di completezza ma anche applicative, come la presente trattazione si proponeva di mettere in luce.

RIFERIMENTI

1. Coelho H., Cotta J.C., Pereira L.M.: 'How to Solve It in PROLOG', Laboratorio Nacional de Engenharia Civil, Lisboa 1982.

2. Covington M.A.: 'Eliminating Unwanted Loops in PROLOG', ACM SIGPLAN Notices, v. 20, n. 1 (1985).

3. Cras J.Y.: 'The Notion of Abstract Data Type in Logic Programming', ESPRIT-ALPES Tech. Rep., (maggio 1984).

4. Goguen G.A., Meseguer J.: 'Equality, Types, Modules and (why not?) Generics for Logic Programming', Jour. Log. Prog., n. 2, pp. 179-210 (1984).

5. Kowalski R.A.: 'Predicate Logic as Programming Language', IFIP 74, North Holland, pp. 569-574 (1974).

6. Nute D.: 'A Programming Solution to Certain Problems with Loops in PROLOG', ACM SIGPLAN Notices v. 20, n. 8 (agosto 1985).

7. Poole D., Goebel R.: 'On Eliminating Loops in PROLOG', ACM SIGPLAN Notices v. 20, n. 8, (agosto 1985).

8. Warren D.: 'Implementing PROLOG', voll. I e II, DAI Res. Rep. 39 e 40, Edimburgh Univ. (1977).

ASPETTI DI UNA RICERCA CSELT SU MACCHINE E LINGUAGGI DI NUOVA GENERAZIONE

P.G.Bosco, G.Giandonato, S.Giorcelli, E.Giovannetti, G.Sofi

CSELT, Centro Studi e Laboratori Telecomunicazioni
via G. Reiss Romoli 274, Torino

Sommario

Nell'articolo sono descritti gli indirizzi di un programma di ricerca su macchine e linguaggi di nuova generazione, fondati sulla programmazione logica. Tali indirizzi riguardano, in particolare: lo sviluppo di un'architettura parallela MIMD basata su un insieme di nodi realizzati ciascuno attorno ad un Transputer a 32 bit, e connessi da una rete multistadio a pacchetto di tipo Delta; lo studio di modelli computativi paralleli per il Prolog appropriati per applicazioni di IA in tempo reale; la definizione di un linguaggio integrato logico e funzionale e di un corrispondente modello esecutivo; l'utilizzo della programmazione logica nel campo dei linguaggi di specifica per la descrizione e il progetto di grossi sistemi di telecomunicazioni.

1. Introduzione

Questo lavoro descrive gli indirizzi di un programma di ricerca attivo da circa due anni in CSELT su macchine e linguaggi cosiddetti di nuova generazione ed attinente dunque per larga parte alle tematiche della programmazione logica. Questa non e' pero' assunta come paradigma iniziale o di riferimento della ricerca che, piuttosto, e' volutamente pilotata dalle applicazioni e tesa ad accertare se e come su problemi specifici ma reali e di grossa taglia (ad es. lo studio sintattico-semantico dei sistemi di comprensione del linguaggio naturale parlato) le promesse accompagnatesi alla rapida crescita d'interesse e di attivita' sull'elaborazione di nuova generazione siano, almeno in parte, mantenibili.

Lo spazio in cui si muove questa attivita' e' dunque quello, tipico per CSELT, a meta' strada fra la ricerca di base, a contenuti prevalentemente teorici come nelle Universita', la cui collaborazione risulta in questo caso e proprio per via di tali contenuti essenziale sugli aspetti piu' avanzati del programma (vedi l'esempio, piu' avanti, di cooperazione nel progetto su linguaggi logico-funzionali), e la ricerca-sviluppo dei laboratori industriali, direttamente mirata ai possibili prodotti. In particolare per i temi affrontati questo spazio e' tuttora ritenuto molto vasto e tale da giustificare approcci pragmatici (come nel caso delle architetture ad elevato parallelismo, vedi par. 2), o una diversificazione degli sforzi sia nel tempo sia nella dimensione tecnica degli obbiettivi (come per i linguaggi di programmazione non-convenzionali, par. 3), o, infine, una fase di studio preliminare (e' il caso dell'applicazione all'ambiente del software di telecomunicazioni di linguaggi formali di specifica, par.4).

2. Architetture parallele

Come e' noto, una delle piu' rilevanti promesse della elaborazione di nuova generazione consiste nella intravista possibilita' di sfruttare l'elevato grado di parallelismo di architetture fisiche quali quelle rese fattibili dallo sviluppo delle tecnologie VLSI mediante stili di programmazione dichiarativi, a semantica non intrinsecamente sequenziale perche' basati su "regole" (di riscrittura, di implicazione logica, di produzione, e di comportamento, come nei linguaggi funzionali, in quelli logici, nei sistemi di inferenza forward, e nei linguaggi di programmazione concorrente tipo CP, rispettivamente). Trattandosi di macchine orientate al linguaggio l'approccio piu' corretto in linea di principio sarebbe quello ("language first") di far discendere il disegno dell'architettura dal modello computazionale e questo dalla semantica operazionale del linguaggio non-convenzionale adottato. In assenza, come nel nostro caso, di un unico linguaggio di riferimento ed anzi essendo obbiettivo della ricerca l'individuazione per un tale linguaggio delle primitive di controllo del parallelismo piu' adatte alla classe di problemi affrontata, l'approccio alla definizione dell'architettura fisica (qui distinta da quella virtuale che su di essa implementa l'esecutore parallelo) e' dal basso ossia e' derivata da vincoli di tipo tecnologico, principalmente legati all'ottica VLSI, e da caratteristiche ritenute comuni alla varieta' dei modelli computativi indagati.

La macchina in progetto, di cui e' previsto un prototipo con grado di parallelismo sufficiente per una realistica sperimentazione (64-128 processori), si basa in particolare sulla eliminazione di memoria comune, sostituita dalla possibilita' di uno schema di indirizzamento globale ad una memoria

omogeneamente distribuita sugli elementi computativi (il motivo e' ovviamente legato all'esigenza di omogeneita', essenziale nella prospettiva tecnologica di lungo termine in cui ogni nodo processore + modulo di memoria + elemento di comunicazione sara' realizzabile con pochissimi circuiti); sul privilegiamento di modelli computazionali con granularita' media (ossia tali da allocare in sequenza sul singolo processore una quantita' significativa di istruzioni da eseguire per ogni unita' di lavoro schedulata; qui il motivo e' rintracciabile nella necessita' di evitare eccessivi overheads nello sfruttamento del parallelismo e nell'obbiettivo di assicurare su ogni nodo del sistema parallelo l'efficienza ottenuta sul processore sequenziale mediante le tecniche convenzionali, ad es. compilative, disponibili per esecutori control-flow); su uno schema di comunicazione interprocessor, basato su commutazione veloce di pacchetto, consistente con i requisiti di scalabilita' (scegliendo opportune topologie di rete), relativa insensibilita' ai ritardi (mediante elementi computativi caratterizzati da rapidissimo context-switching) e ottimizzazione della localita'.

Il prototipo target e' costituito da un insieme omogeneo di nodi costruiti ciascuno attorno ad un Transputer a 32 bit e connessi da una rete multistadio a pacchetto di tipo Delta, realizzata mediante un circuito integrato custom, attualmente in fase avanzata di sviluppo presso CSELT. Una rete statica, realizzata direttamente con i link disponibili sul Transputer, permette l'ottimizzazione, ad es. ai fini del bilanciamento di carico, della comunicazione strettamente locale, di tipo nearest-neighbours, ed il raccordo con l'elaboratore host, un Microvax II, per caricamento, collezione dei risultati e monitoraggio del sistema. Nella nostra ricerca il Transputer e' considerato come una "approssimazione" dell'elemento computativo ideale, e tra gli obbiettivi collaterali del programma si colloca anche quello di derivare le caratteristiche ulteriori che un processore di questo tipo (ossia RISC, con comunicazione e multitasking built-in, a veloce process-switching) dovrebbe possedere per risultare ottimale nelle applicazioni di elaborazione simbolica.

3. Linguaggi di programmazione

Un tipico problema che richiede un elevato grado di parallelismo nell'elaborazione simbolica e' il processo di comprensione del linguaggio naturale parlato ai livelli di trattamento sintattico, semantico e pragmatico, dove l'incertezza dei dati in ingresso (nel sistema attualmente in sviluppo presso CSELT un database, proveniente dal precedente stadio di riconoscimento del segnale, di ipotesi lessicali concernenti la probabile presenza di una parola in un intervallo temporale), accompagnata al gran numero di regole e fatti coinvolti

nell'analisi ed all'esigenza di tempo reale imposta dalle possibili applicazioni, genera requisiti di throughput incompatibili con le tecniche elaborative convenzionali. In termini di programmazione logica il problema puo' essere raffrontato con quello, storicamente legato al Prolog fin dalla sua origine (vedi DCG e simili), della analisi di linguaggio naturale, con la differenza essenziale pero' che nel caso del parlato sono richieste strategie di controllo nella generazione e nell'esplorazione dello spazio di ricerca ben piu' complesse (ad es. parsificazione non left-to-rigth, mista top-down e bottom-up, con memorizzazione dei risultati intermedi, guidata da euristiche programmabili del tipo best-first, etc.).

Il punto di partenza, e di riferimento per accertare i vantaggi di un approccio basato sulla programmazione logica, e' l'implementazione di una versione parallela degli algoritmi di comprensione mediante un ambiente di processi LISP concorrenti e comunicanti a messaggi asincroni. Gia' in tale ambiente, che e' in corso di sviluppo sulla macchina sopra descritta, sono stati notati i vantaggi espressivi che deriverebbero dall'aver integrata nella parte funzionale del LISP un'efficiente componente logica interfacciata al database di fatti e regole che costituiscono la base di conoscenza del singolo processo.

L'obbiettivo e' la riformulazione in termini di programmazione logica degli stessi algoritmi al fine di una valutazione comparata dei benefici e degli svantaggi dei due approcci. Appare evidente, dagli accenni di cui sopra, come estensioni del Prolog, contemporaneamente adatte alla specificita' del problema e ad un'efficiente realizzazione parallela, siano necessarie e come d'altra parte esse non siano banali, sia dal punto di vista concettuale sia implementativo (si pensi ad una "assert" in architetture distribuite).

Un primo passo della ricerca e' consistito nell'analisi e nell'organizzazione il piu' sistematica possibile dei modelli computativi paralleli per il Prolog. Quattro modelli sono stati scelti fra i piu' significativi proposti recentemente in letteratura (quello di Conery, di Haridi e Ciepielewski, di Lindstrom, ed il PIE) e lo sforzo e' stato quello di confrontarli e correlarli in modo da ottenere un quadro generale dei problemi da affrontare e delle tecniche che si possono utilizzare per l'implementazione di un linguaggio logico. In particolare sono stati esaminati a fondo i seguenti aspetti:

- forma e grado di parallelismo. Come e' noto, Conery ha distinto quattro forme di parallelismo (OR, AND, stream, search), ma in realta' questa classificazione risulta grossolana per i nostri scopi; ad es. all'interno della forma OR diversi gradi di parallelismo sono possibili ("pipelining", "backtracking free", ecc.), corrispondentemente a differenti primitive di controllo richieste come estensione al linguaggio.

- rappresentazione dei dati. Dal nostro studio e' emerso che il classico problema

copying/sharing ha tre diversi aspetti, che riguardano rispettivamente la rappresentazione di "goal", "skeleton" ed "environment". In particolare in un ambiente parallelo e' di importanza vitale trovare una soluzione accettabile per l'ultimo aspetto (environment copying/sharing). Una soluzione ottimale e' strettamente dipendente non solo dall'architettura fisica (ad es. memoria centralizzata oppure distribuita), ma anche dal tipo di applicazione. Comunque sembrano attraenti, in quanto buon compromesso, strategie miste basate sul "copying incrementale" ed il "lazy fetch".

- conflitto "OR". Un problema tipico nell'implementazione del Prolog e' dato dal cosiddetto "conflitto OR", strettamente connesso al nondeterminismo "don't know" del linguaggio. Un interprete sequenziale risolve tale conflitto grazie ai concetti di: unico "binding environment", attraversamento in profondita' dell'albero di ricerca, procedura di "backtracking", "trail list". Per un'esecuzione OR-parallela invece sono stati introdotti altri metodi, riconducibili a due tipi di meccanismi base: replica ad ogni nodo nondeterministico delle variabili ancora non istanziate (si vedano ad es. il "binding array" di D.S. Warren e la recente implementazione del CP di Tacheuchi); "unificazione in due fasi" (Wise, Lindstrom), secondo la quale parte delle assegnazioni dovute all'unificazione (e precisamente quelle relative alle variabili del goal corrente) non sono eseguite all'atto della chiamata della clausola, ma rinviate alla seconda fase, cioe' quando la clausola e' conclusa ed il controllo ritorna al goal corrente. La scelta fra questi due metodi dipende fortemente dal tipo e dalla forma di parallelismo che si intende supportare ed inoltre e' in stretta correlazione con la questione copying/sharing illustrata precedentemente.

Un tema di ricerca connesso con i precedenti, ma piu' a lungo termine, e motivato piu' in generale dalla necessita', in molte applicazioni, di tipi diversi di elaborazione, cioe' di una parte procedurale/algoritmica, tipicamente deterministica, e di una parte dichiarativa/inferenziale, tipicamente nondeterministica, le quali sono molto naturalmente esprimibili rispettivamente in un linguaggio funzionale e in un linguaggio logico, e' quello che viene affrontato dallo CSELT, unitamente all' Universita' di Pisa, all'interno di un progetto ESPRIT nel sottoprogetto "Integration of logic and functional languages", che ha come obiettivi dapprima la definizione di un linguaggio unitario in cui i due stili di programmazione rispettivamente logico e funzionale siano integrati in modo "profondo", cioe' con una semantica chiara e ben definita, in contrapposizione al semplice interfacciamento di due linguaggi distinti; poi la definizione di un modello computazionale che tratti anch'esso in modo unitario i due aspetti del linguaggio, e che contemporaneamente sfrutti l'elevato grado di parallelismo che, come si e' detto, e' oggi possibile realizzare su un'architettura fisica.

L'attivita' in CSELT in questo primo anno del progetto e' stata rivolta soprattutto all'analisi del significato dell'integrazione logico-funzionale dal punto di vista della logica del prim'ordine con uguaglianza, in particolare della

logica delle clausole di Horn con uguaglianza. Tale indagine ha messo in evidenza, da un lato, la continuita' concettuale dai sistemi di inferenza generali per la logica con uguaglianza (ad esempio risoluzione piu' paramodulazione) fino ai sistemi inferenziali specializzati adottabili come interpreti di linguaggi di programmazione. D'altro lato ha posto in risalto proprio il fatto che sistemi generali, utilizzabili come dimostratori di teoremi, sono invece poco adatti a servire di base per interpreti di veri linguaggi di programmazione, in cui si vuole che la computazione non sia semplicemente ricerca in uno spazio di possibili soluzioni, ma resti invece ancora in qualche modo sotto il controllo del programmatore. Piu' in concreto, si tratta dell'esigenza che l'algoritmo di inferenza sia "lineare" nello stesso senso in cui lo e' la risoluzione SLD che sta alla base del Prolog; cio' equivale a richiedere che si prendano in considerazione soltanto classi di "programmi" per le quali l'algoritmo di soluzione di goals fondato su risoluzione lineare e narrowing - o, equivalentemente, solo sulla risoluzione, previo "appiattimento" del programma - sia completo. Si esclude cosi' il ricorso ad algoritmi di completamento alla Knuth-Bendix e simili, tranne eventualmente in una fase di "compilazione", di cui pero' non e' in generale garantita la terminazione, data la natura semidecidibile del problema.

Uno dei principali nodi da affrontare a livello di definizione del linguaggio e' costituito dall'integrazione nella cornice di cui sopra delle funzioni di ordine superiore, che costituiscono una caratteristica essenziale dei linguaggi funzionali moderni. A questo riguardo un'attraente direzione potrebbe risultare quella indicata dai lavori di Martin-Löf e di molti altri che si fonda sulla matematica costruttiva e sulla logica intuizionistica.

4. Linguaggi di specifica

Da anni lo CSELT e' attivo nel campo dei linguaggi di specifica per la descrizione e il progetto di complessi sistemi di telecomunicazioni. Una intensa attivita' si e' sviluppata in ambito C.C.I.T.T. per la definizione di un linguaggio di specifica (SDL) capace di trattare aspetti sequenziali, concorrenti e di strutturazione; parallelamente in CSELT si porta avanti una ricerca di maggiore generalita' su tali temi e con maggiore enfasi sui tools di supporto a tali linguaggi.

Dal punto di vista della definizione di linguaggi di specifica l'aspetto logico ha fatto il suo ingresso a livello C.C.I.T.T. nella formulazione della nuova proposta Z104 (la componente "abstract data type" dell'SDL) che con un forte contributo di CSELT-SIP e' stata indirizzata verso una forma di clausole di Horn

con uguaglianza (tipo EQLOG). Si prevede che tale aspetto sara' ulteriormente enfatizzato nel periodo successivo (Questione II).

Riguardo all'utilizzo specifico di programmazione logica, si puo' dire che in generale si utilizza il Prolog come linguaggio di manipolazione simbolica, in alternativa al LISP, in quelle attivita' in cui nondeterminismo e unificazione sono elementi caratteristici del problema. In particolare:

a) Definizione di semantiche operazionali con regole d'inferenza ('a la Plotkin') dei linguaggi di specifica esistenti. Con questo metodo si riescono ad ottenere in breve tempo dei simulatori piu' o meno simbolici (graph builders) per i linguaggi d'interesse. Il meccanismo di backtracking semplifica notevolmente l'analisi del nondeterminismo don't care presente nelle parti concorrenti.

b) Implementazione di procedure di decisione per logiche particolari (ad esempio temporali). Il Prolog viene utilizzato per l'implementazione delle componenti "speciali" delle logiche in questione, ad esempio attraverso l'esplicitazione della loro definizione semantica tramite predicati del tipo di "holds". La componente logica del prim'ordine viene in genere demandata al Prolog stesso.

c) Strumenti di supporto alla specifica con Abstract Data Types. Il Prolog viene utilizzato con l'ottica "logica" nella conversione di regole di riscrittura in clausole che assiomatizzano un certo predicato di uguaglianza come linguaggio di programmazione per l'implementazione di procedure di riscrittura, narrowing, completamento. In taluni casi ci si scontra con l'assenza dell'occur check (ad esempio nel calcolo delle coppie critiche nell'algoritmo di Knuth-Bendix), che deve pertanto essere programmato in Prolog con ovvie inefficienze Vengono anche implementati algoritmi speciali di unificazione (C, AC) che hanno come base l'unificazione standard. Il Prolog fornisce gia' un ambiente in cui non ci si deve preoccupare della rappresentazione dei bindings, delle varibili, etc.

d) Fast Prototyping, sperimentazione di nuovi linguaggi e integrazione di aspetti funzionali e logici. Un interessante esperimento ha condotto alla costruzione di un mini-ambiente funzionale (tipo ML). L'insieme di type-checker e traduttore che e' stato sviluppato in tre giorni e ammonta a circa 5 pagine Prolog e' in grado di trattare "efficientemente" gli oggetti funzionali sfruttando l'idea originale di Warren.

5. Note conclusive

Naturalmente l'esposizione qui riportata non e' esaustiva delle applicazioni o degli interessi connessi alla programmazione logica presenti in CSELT, dove esperienze significative, ad es. nel campo dei sistemi esperti, sono state maturate, ed altre sono in corso.

Il lavoro descritto e' parzialmente finanziato dai progetti ESPRIT N.26 (Advanced Algorithms and Architectures for Signal Recognition and Understanding) cui collaborano gruppi del Dipartimento di Informatica dell'Universita' di Torino, e N.415 (Parallel Architectures and Languages for Advanced Information Processing), al quale lo CSELT partecipa in cooperazione con il Dipartimento di Informatica dell'Universita' di Pisa.

VERSO UN AMBIENTE DI PROGRAMMAZIONE IN PROLOG

A.Martelli e G.Rossi

Dipartimento di Informatica, Universita' di Torino
Via Valperga Caluso, 37 - 10125 TORINO

SOMMARIO

In questo lavoro vengono presentati alcuni strumenti di base di un ambiente di programmazione Prolog in corso di realizzazione presso il Dipartimento di Informatica di Torino. In particolare viene descritto l'interprete Prolog realizzato, mettendo in evidenza l'approccio seguito nella sua realizzazione, con particolare riferimento alla definizione per passi successivi della semantica operazionale da cui questo strumento viene derivato. Vengono quindi descritte le caratteristiche principali di altri due strumenti di base dell'ambiente: un editor, orientato ai costrutti del linguaggio ed un interprete-debugger, entrambi scritti in Prolog. Viene inoltre fatto cenno ad altri strumenti di programmazione progettati o realizzati ed al problema dell'integrazione di tutti gli strumenti mediante la definizione di una comune rappresentazione interna dei programmi.

1. INTRODUZIONE

Scopo principale di questo progetto e' la realizzazione di un ambiente di programmazione avente il duplice obiettivo di fornire un insieme di strumenti per facilitare lo sviluppo di programmi Prolog e di consentire una sperimentazione di estensioni e modifiche del linguaggio stesso.

Relativamente scarsi sono stati fino ad oggi i progetti di ambienti Prolog, specialmente se confrontati ai numerosi sforzi rivolti ad ambienti di programmazione per linguaggi piu' tradizionali, sia di tipo imperativo che funzionale. Alcuni lavori su ambienti di programmazione Prolog sono stati presentati di recente al Congresso su "Languages Issues in Programming Environments" [1] [2], mentre sforzi precedenti in analoga direzione sono riscontrabili nella definizione di estensioni del Prolog, come M_Prolog, KR_Prolog e Micro_Prolog.
L'esperienza maturata nella costruzione di ambienti Lisp puo' comunque essere molto utile anche nel caso del Prolog, sia perche' molti ambienti Lisp sono in uso ormai da anni e contengono una raccolta molto ampia di strumenti, ma soprattutto per la tecnica con cui questi strumenti sono realizzati. Infatti la maggior parte degli strumenti di un ambiente Lisp sono realizzati nello stesso linguaggio Lisp, grazie alla proprieta' di questo linguaggio di trattare i programmi come dati. Anche se il Prolog non possiede questa proprieta', tuttavia le sue capacita' di elaborazione simbolica (come l'unificazione), nonche' alcune primitive meta (come ad es. la "clause" e la "call", oppure quelle per decomporre o costruire un termine) che il Prolog di solito fornisce, consentono di scrivere programmi che manipolano altri programmi (in particolare, interpreti del linguaggio stesso) in modo relativamente semplice, con tecniche di programmazione analoghe a quelle del Lisp.

Il Prolog possiede comunque anche altre caratteristiche, condivise dal Lisp, molto utili alla costruzione di un ambiente di programmazione. In particolare, la presenza di binding dinamico in Prolog facilita l'interattivita' dell'ambiente e la possibilita' di programmazione incrementale, mentre la possibilita' di linking dinamico di procedure definite dall'utente, fa si che un ambiente Prolog risulti facilmente estendibile ed adattabile (ambiente aperto).

Un altro aspetto rilevante di un ambiente di programmazione e' l'integrazione fra i suoi strumenti, proprieta' che consente ad un utente di passare da uno strumento all'altro senza uscire dall'ambiente stesso. Per garantire questo, nel nostro ambiente, tutti gli strumenti opereranno sulla stessa rappresentazione interna dei programmi.

2. SEMANTICA ED INTERPRETE

Punto di partenza e componente basilare di un tale ambiente e' un interprete efficiente per il Prolog. Invece che basarci su un interprete gia' esistente si e' preferito implementarne uno completamente nuovo. Questa scelta e' stata fatta essenzialmente per due motivi: (1) permettere l'utilizzo all'interno del nuovo interprete di un algoritmo di unificazione da noi definito [3], in grado di offrire buone prestazioni e di trattare il caso di termini infiniti; (2) acquisire una conoscenza ed una padronanza della struttura interna dell'interprete stesso, indispensabile per poterlo successivamente integrare con il resto degli strumenti dell'ambiente, e che difficilmente si riesce ad ottenere con la semplice analisi di un interprete gia' esistente.

In primo luogo e' necessario riformulare l'algoritmo di unificazione presentato in [3] in una forma piu' orientata ad una sua implementazione concreta. Nella sua formulazione piu' astratta, direttamente derivata da quella proposta in [4], l'algoritmo opera su un sistema di multiequazioni, a cui vengono applicate, in modo non-deterministico, una serie di trasformazioni fino a che il sistema non assume una forma particolare, detta forma risolta, da cui e' immediato ricavare l'mgu dei termini dati. Da questa formulazione dell'algoritmo, ne e' stata ricavata una equivalente ma deterministica e tale da poter essere facilmente implementata con un linguaggio imperativo tipo Pascal, utilizzando strutture dati dinamiche con puntatori. In pratica, un sistema di multiequazioni puo' essere implementato come una struttura dati, eventualmente ciclica, in cui ciascun nome di variabile e' interpretato come l'indirizzo di una cella di memoria contenente il legame della variabile stessa. La rappresentazione in memoria dei termini puo' poi essere data in modo tale da permettere che la costruzione di nuovi termini, richiesta dall'algoritmo di unificazione utilizzato, possa avvenire modificando quelli gia' presenti, senza richiedere l'allocazione di nuova memoria. Visto in quest'ottica, l'algoritmo di unificazione da noi utilizzato viene ad essere molto simile ad altri algoritmi di unificazione per termini infiniti, quali quello di Fages [5] e quello di Colmerauer [7]. Una caratteristica peculiare del nostro algoritmo rimane comunque una maggior efficienza nel caso di unificazione di termini complessi (cioe' termini contenenti altri termini non variabili ne costanti).

Il passo successivo e' stato quello di vedere l'algoritmo di unificazione

proposto inserito nel contesto dell'interprete Prolog. Punto di partenza e' la definizione astratta di computazione di un programma logico data da Colmerauer [6] ed in grado di trattare sia termini finiti che infiniti, opportunamente adattata al nostro algoritmo. Secondo questa definizione, una computazione logica puo' vedersi in modo astratto come una successione di stati ciascuno caratterizzato da un sistema di multiequazioni, che contiene i legami fino a quel punto costruiti e da un insieme di goal ancora da soddisfare. Ad ogni passo della computazione si aggiunge una nuova multiequazione al sistema, costituita dal goal corrente e dalla testa della clausola selezionata e si risolve il nuovo sistema cosi' ottenuto. Il procedimento e' ripetuto fino a che non ci sono piu' goal da soddisfare.

Partendo da questa formulazione piu' astratta della semantica operazionale di una computazione logica, e' possibile giungere a formulazione sempre piu' concrete, fino ad arrivare ad una definizione che sia facilmente traducibile in un interprete reale [8]. E' questo l'approccio che si e' seguito nella definizione dell'interprete Prolog da noi utilizzato.

Una prima riformulazione piu' precisa del significato di computazione logica e' mostrata in Fig. 1 nella forma di un programma logico, con la usuale sintassi Prolog.

```
prove([], DB, FSys, FSys).
prove([Goal|GList], DB, Sys, RSys):-
    clauses(Goal, DB, SClauses),
    try(Goal, SClauses, DB, Sys, NewSys),
    prove(GList, DB, NewSys, RSys).

try(Goal, [Clause|CList], DB, Sys, RSys):-
    rename(Clause, Sys, clause(Head,Body), NSys),
    unify(Head, Goal, NSys, NewSys),
    prove(Body, DB, NewSys, RSys).
try(Goal, [Clause|CList], DB, Sys, RSys):-
    try(Goal, CList, DB, Sys, RSys).
```

- Figura 1 -

Questo programma ha la struttura tipica di un interprete Prolog, con due procedure, "prove" e "try", che si richiamano ricorsivamente l'un l'altra. La prima scandisce la lista dei goal da dimostrare, fino a che questa non e' vuota. La seconda, per ogni goal estratto dalla lista, scandisce le clausole del database di programma DB, cercandone una la cui testa Head unifichi con il goal dato, Goal. La scelta della clausola in DB e' fatta in modo non-deterministico (non-determinismo OR), a seconda di quale alternativa "try" viene selezionata. Gli eventuali sottogoal del corpo (Body) della clausola trovata vengono dimostrati richiamando nuovamente la procedura "prove".
Il sistema di multiequazioni Sys viene passato dalla "prove" alla "try" che lo modifica inserendovi le nuove multiequazioni ottenute dall'unificazione del goal dato con la clausola selezionata (predicato "unify"), dopo aver opportunamente rinominato le variabili di quest'ultima (predicato "rename").
Il predicato "clauses" determina l'insieme delle clausole del DB aventi lo stesso nome del predicato in Goal e permette percio' di ridurre il numero di tentativi di chiamate ad "unify".

L'interprete di Fig. 1 puo' essere ulteriormente modificato per tener conto dell'idea di "environment" (env) tipica dei linguaggi di programmazione convenzionali. Nel nostro caso, l'env servira' per associare gli identificatori delle variabili contenute in un termine sorgente ai nomi interni, consentendo cosi' di eseguire il "renaming" di ogni singolo goal, piuttosto che dell'intera clausola, soltanto quando necessario. La chiamata della "rename" viene percio' sostituita dalla sequenza

```
mkenv(V,Sys,NSys,NewEnv),
copy(Head,NewEnv,CHead), ...
```

dove V e' l'insieme delle variabili contenute nella clausola in esame e CHead e' il nuovo termine ottenuto rinominando le variabili di Head in accordo con il nuovo ambiente NewEnv.

Questo permette tra l'altro di meglio evidenziare le analogie con gli interpreti per linguaggi di programmazione convenzionali. L'operazione di renaming e' analoga all'operazione di valutazione, in un dato env, dei parametri attuali nella chiamata di una procedura in un linguaggio convenzionale. Il concetto di sistema di equazioni invece e' analogo al concetto di store, tipico della semantica dei linguaggi imperativi. Come lo store, infatti, un sistema di equazioni e' un oggetto globale e modificabile, al contrario dell'env. Lo store, in particolare, conterra' i dati e cioe' i termini costruiti, tramite "renaming", a partire dal codice e cioe' dai termini sorgenti contenuti nelle clausole.

Applicando opportune trasformazioni e' possibile ottenere un interprete ricorsivo deterministico a partire da quelli sopra descritti. Una possibile definizione di un tale interprete e' mostrata in Fig. 2. Rispetto alla versione di Fig. 1, il nuovo interprete ha due argomenti in piu', AR e BT. AR e' uno stack di record di attivazione, ciascuno costituito dal termine

```
ar(GList,Env)
```

dove GList e' la lista dei goal ancora da provare e Env e l'ambiente attuale. BT invece e' uno stack di record di backtracking, ciascuno della forma

```
bt(Goal,Clist,Sys,AR)
```

e permette di ripristinare lo stato precedente in caso di backtracking. La procedura "prove" ammette altre due alternative rispetto a quella data in Fig. 1: (a) non ci sono piu' goal da dimostrare, ma AR non e' ancora vuoto e quindi e' necessario svuotarlo, e (b) l'unificazione e' fallita, restituendo come risultato il termine err invece che sys(Sys), come avviene nel caso corretto, e quindi e' necessario richiamare la "try", ripristinando la stato contenuto in BT.

```
prove(GList,DB,Env,err,AR,[bt(OGoal,OCList,OSys,OAR)|BT],RSys):-
    try(OGoal,OCList,DB,OSys,OAR,BT,RSys).
prove([],DB,Env,sys(FSys),[],BT,sys(FSys)).
prove([],DB,Env,sys(Sys),[ar(GList,NEnv)|AR],BT,RSys):-
    prove(GList,DB,NEnv,sys(Sys),AR,BT,RSys).
prove([Goal|GList],DB,Env,sys(Sys),AR,BT,RSys):-
    clauses(Goal,DB,SClauses),
    copy(Goal,Env,CGoal),
    try(CGoal,SClauses,DB,sys(Sys),[ar(GList,Env)|AR],BT,RSys).

try(Goal,[],DB,Sys,AR,[bt(OGoal,OCList,OSys,OAR)|BT],RSys):-
    try(OGoal,OCList,DB,OSys,OAR,BT,RSys).
```



```
try(Goal,[clause(V,Head,Body)|CList],DB,Sys,AR,BT,RSys):-
    mkenv(V,Sys,NSys,NewEnv),
    copy(Head,NewEnv,CHead),
    unify(Goal,CHead,NSys,NewSys),
    prove(Body,DB,NewEnv,NewSys,AR,[bt(Goal,CList,Sys,AR)|BT],RSys).
```

Figura 2. Interprete deterministico

Questo interprete e' in una forma adatta ad essere facilmente tradotta in una implementazione concreta, scritta con un linguaggio imperativo tradizionale (Pascal, C, ...). Ovviamente per ottenere una implementazione efficiente e' necessario introdurre varie ottimizzazioni e seguire opportune tecniche di implementazione. Una possibile ottimizzazione riguarda la possibilita' di ridurre la quantita' di dati copiati dall'interprete di Fig. 2. Questo si puo' ottenere fondendo la "copy" e la "unify" in un unico predicato i cui argomenti sono ora due coppie, ciascuna costituita dal termine da unificare e dal relativo env (e cioe' una "chiusura"). In questo modo i termini sorgenti possono essere copiati soltanto se e quando inseriti nello "store", cosi' come avviene solitamente con la tecnica di "structure copying". L'uso delle chiusure puo' poi essere esteso permettendo che anche i termini costuiti siano rappresenti in questo modo, secondo la nota tecnica di "structure sharing". Vantaggi e svantaggi delle due tecniche sono ampiamente discussi nella letteratura (si veda ad es. [9]). Qui si puo' osservare in piu' che la tecnica di "structure sharing" poco si presta ad essere impiegata con un algoritmo di unificazione che richiede la creazione di nuovi termini, come ad es. la "common part" nell'algoritmo in [4]. Un'altra caratteristica di una implementazione reale e' l'utilizzo di un terzo stack, normalmente detto stack di trail, in cui memorizzare lo stato delle variabili prima che questo venga modificato da un'operazione di unificazione, invece che inserire in BT l'intero sistema, ad ogni punto di backtracking.

Attualmente, sono state realizzate alcune implementazioni prototipali di un nucleo di un interprete Prolog avente la struttura fin qui descritta. Inoltre e' in corso una nuova reimplementazione di questi prototipi, scritta in linguaggio C, mirante ad ottenere un prodotto piu' completo ed efficiente. Rimangono comunque inalterate le scelte sull'algoritmo di unificazione e sulle tecniche di implementazione utilizzate in precedenza.

Si osservi che, l'approccio seguito, in cui si e' cercato di evidenziare le analogie con i linguaggi di programmazione convenzionali, ha il duplice vantaggio di permettere l'utilizzazione di tecniche di implementazione analoghe a quelle tradizionali, ma anche di fornire una solida struttura per estendere i linguaggi di programmazione logica nella direzione dei linguaggi di programmazione convenzionali.

3. EDITOR E DEBUGGER

Oltre alla definizione ed implementazione dell'interprete si e' iniziato nel contempo lo sviluppo di altri strumenti di base dell'ambiente di programmazione che si intende realizzare. Tutti gli strumenti dell'ambiente sono scritti completamente in Prolog e quindi si appoggiano all'interprete Prolog realizzato. Si e' comunque deciso di estendere il Prolog con nuovi predicati meta, direttamente supportati dall'interprete, ogni volta che per ragioni di

efficienza o comodita' questo risultava piu' appropriato.

L' esperienza acquisita in questi ultimi anni con lo sviluppo di ambienti di programmazione per linguaggi tradizionali imperativi (Pascal, Ada, ecc.) mostra che uno degli aspetti principali di questi ambienti e' che tutti gli strumenti operino sulla stessa rappresentazione interna dei programmi. Il tipo di rappresentazione adottata e' generalmente una struttura ad albero che riproduce la struttura sintattica del programma relativamente alla cosiddetta sintassi astratta, ossia la sintassi che descrive i costrutti di un linguaggio di rilevanza semantica senza tener conto dei problemi legati alla rappresentazione esterna dei programmi come testo ("zucchero sintattico"). In particolare l'interazione fra l'utente e l'ambiente viene normalmente realizzata con un editor guidato dalla sintassi, che consente di costruire direttamente programmi nella sopra menzionata rappresentazione interna.

Nel caso del Prolog, la sintassi del linguaggio risulta particolarmente semplice e non si e' ritenuto utile sviluppare un vero e proprio editor guidato dalla sintassi. Si e' anzi ritenuto che il tipo di interazione imposto da questi editor risulti inutilmente limitativa ed inefficiente nel caso del Prolog. Cio' nonostante, l'editor definito offre varie funzionalita' tipiche di un editor guidato dalla sintassi, oltre a funzioni di editing generali. In particolare, l'editor fornisce vari comandi orientati ai costrutti del linguaggio, come la possibilita' di operare su predicati e non su caratteri, spostendosi ad es. da un predicato al successivo, o sostituendo il nome di tutti i predicati componenti una data procedura, ecc..

Inoltre l'editor memorizza i programmi in una forma interna particolare, utilizzata poi da tutti gli altri strumenti dell'ambiente. Ciascuna clausola letta e sintatticamente corretta viene memorizzata dall'editor sotto forma di un predicato Prolog, contenente le opportune informazioni, necessarie agli altri strumenti dell'ambiente. Un punto critico per definire la rappresentazione interna e' proprio stabilire quali devono essere queste informazioni. Informazioni utili sono senz'altro le seguenti:

- numero e nome simbolico delle variabili di una clausola;
- riferimento alla clausola precedente e a quella successiva;
- numero d'ordine della clausola;
- informazioni per il debugger.

Alcune di queste informazioni, come ad es. il numero e il nome simbolico delle varibili potrebbero essere mantenute direttamente dall'interprete Prolog e rese disponibili tramite opportuni predicati di sistema. Viceversa, potrebbero essere mantenute soltanto nella rappresentazione interna e quindi essere ignorate dall'interprete Prolog. La soluzione da noi seguita e' una soluzione di compromesso tra queste due possibilita'.

Attualmente, un editor con le caratteristiche sopra delineate e' in fase di realizzazione. E' in corso di realizzazione anche un interprete-debugger per il Prolog tale da poter essere integrato con il resto degli strumenti. L'approccio seguito nella definizione di questo strumento e' stato quello di partire dall'interprete di Fig. 2 e di estenderlo opportunamente. Questo significa, innanzitutto, renderlo sufficientemente efficiente nell'eseguire programmi scritti nella rappresentazione interna generata dall'editor, e quindi inservi opportuni controlli in modo da poter individuare piu' precisamente lo stato della computazione in corso. Questo approccio permette di definire il

debugger sulla base di un preciso modello di computazione logica, che e' lo stesso da cui e' stato derivato l'interprete Prolog, e a cui l'utente puo' fare riferimento nell'interpretare le informazioni che il debugger gli fornisce (cfr. [1]). Il debugger sara' comunque dotato di funzioni di debugging abbastanza tipiche, come quelle descritte ad es. in [10].

Infine, si prevede anche che l'editor realizzato sara' successivamente esteso dotandolo di semplici capacita' di analisi e ragionamento sui programmi Prolog che esso gestisce, in modo analogo all'editor descritto in [2].

4. ALTRI STRUMENTI

Sono stati anche gia' realizzati o sono in corso di realizzazione vari altri strumenti che potrebbero andare ad arricchire il nostro ambiente di programmazione.

Tra essi, uno strumento di notevole importanza e' un "type checker" per il Prolog. E' esperienza comune per chi programma in Prolog di avere la risposta "no" dall' interprete ai primi tentativi di eseguire un programma, e di scoprire poi con fatica, mediante l'uso degli strumenti di traccia e di debugging che l'errore era dovuto alla scrittura sbagliata del nome di qualche predicato o funzione oppure all' uso di un predicato con gli argomenti di tipo sbagliato (ad es. passando un solo termine dove invece ci si aspettava una lista). Uno strumento potente ed efficace per scoprire errori di questo tipo e' costituito dalle regole dei tipi e dall'obbligo delle dichiarazioni nei linguaggi tradizionali.

Sembra quindi opportuno prevedere di estendere anche i linguaggi logici con una struttura dei tipi, in modo che il programmatore possa, opzionalmente, associare delle dichiarazioni di tipo a tutti i predicati ed alle funzioni usati in un programma. In questo modo un programma di "type checking" sara' in grado di verificare la correttezza del programma Prolog rispetto alle regole dei tipi. E' stato mostrato che le possibilita' di pattern matching del Prolog consentono di avere una struttura dei tipi molto ricca, simile a quella del linguaggio ML, che infatti fa uso dell' algoritmo di unificazione per la verifica dei tipi. Come si e' gia' constatato per i linguaggi tradizionali, anche un "type checker" per il Prolog non e' altro che un interprete particolare che opera su un dominio non standard, quello dei tipi, invece che sui valori usuali.

Si noti che per eseguire il "type checking" e' necessario valutare tutti i termini, mentre questa operazione non viene eseguita nel normale modo di esecuzione. Puo' essere conveniente quindi per realizzare l'interprete per il type checking di disporre di un altro strumento: un interprete che valuti tutti i termini. Questo strumento e' utile anche di per se, consentendo la estensione del Prolog nella direzione di una integrazione con i linguaggi funzionali.

Brevemente, altri strumenti gia' realizzati al momento attuale sono:

- alcuni preprocessor, utilizzati sia per definire in modo semplice estensioni al Prolog, come la possiblita' di avere una notazione funzionale o di definire macro [11], sia per "nascondere" dettagli di implementazione all'utente, in problemi di rappresentazione della conoscenza [12].

- nucleo di un interprete per il "Concurrent Prolog";

Un problema ancora aperto e' l'integrazione di questi strumenti, sviluppati per diversi scopi, con il resto dell'ambiente sopra descritto.

4. RIFERIMENTI BIBLIOGRAFICI

[1] N.Francez et al.: An Environment for Logic Programming; in Proc. of the ACM SIGPLAN 85 Symp. on Languages Issues in Programming Environments; Seattle, Washington, 25-28 June 1985, 179-190.

[2] H.J.Komorowski, S.Omori: A Model and an Implementation of a Logic Programming Environment; in Proc. of the ACM SIGPLAN 85 Symp. on Languages Issues in Programming Environments; Seattle, Washington, 25-28 June 1985, 191-198.

[3] A.Martelli e G.Rossi: Efficient Unification with Infinite Terms in logic Programming, in Proc.of the Int. Conf. on 5th. Generation Computer Systems 1984, Tokyo, Japan, Nov. 6-9, 1984.

[4] Martelli, A. and Montanari, U. An Efficient Unification Algorithm; ACM TOPLAS, 4, 2 (April 1982).

[5] Fages, F. Formes canoniques dans les algebres booleennes et applications a la demonstration automatique; These de 3eme cycle, Universite Paris VI, June 1983.

[6] Colmerauer, A. Prolog and Infinite Trees; Logic Programming (K.L. Clark and S-A. Tarnlund eds.), Academic Press, 1982.

[7] Colmerauer, A. Equations and Inequations on Finite and Infinite Trees; Proc.of the Int. Conf. on 5th. Generation Computer Systems 1984, Tokyo, Japan, Nov. 6-9, 1984.

[8] Martelli, A. and Rossi, G. On the Semantics of Logic Programming Languages; Rapporto Interno, Dip. di Informatica, Univ. di Torino, Novembre 1985.

[9] Mellish, C.S. An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter; in Logic Programming (K.L. Clark S-A. Tarnlund eds.), Academic Press, 1982, 99-106.

[10] Eisenstandt, M. A Powerful PROLOG Trace Package, Proc. of ECAI-84, Pisa, Sept. 5-7, 1984, pp. 515-524.

[11] Eggert, P.R. and Val Schorre, D.: Logic enhancement: a method for extending logic programming languages; in Proc. of the ACM Conf. on Lisp and Functional Programming Languages, August, 1982.

[12] Console, L. and Rossi, G.: Implementing Inference Strategies in Prolog based Expert Systems; to be presented at the 8th European Meeting on Cybernetics and System Research, Wien 1986

L'APPROCCIO DELLA METAPROGRAMMAZIONE NEL PROGETTO EPSILON

Paola Franceschi

Systems & Management SpA
Vicolo S. Pierino, 4 - PISA

Carlandrea Simonelli (•)

LIST Srl
Piazza Mazzini, 6 - PISA

ABSTRACT

Uno degli obiettivi primari che il progetto Epsilon si propone di perseguire riguarda l'impiego della tecnica della metaprogrammazione per risolvere in modo uniforme alcuni dei problemi che sorgono nella definizione di un Sistema di Gestione di Basi di Conoscenza (KBMS) come integrazione di due tecnologie: Programmazione Logica e Basi di Dati.

La prima parte del progetto affronta la definizione del linguaggio di rappresentazione della conoscenza. In particolare sono considerate le caratteristiche che un linguaggio logico deve possedere per definire sistemi esperti evoluti: efficienza del processo di inferenza, meccanismi di strutturazione della conoscenza e strumenti di supporto alla definizione e all'utilizzo di sistemi esperti.

1. Una panoramica del progetto Epsilon

In questo lavoro vengono descritte le principali linee di ricerca del progetto Epsilon, evidenziando le tematiche sviluppate durante il primo anno di attivita'.

Il progetto, che è inserito nell'area "Advanced Information Processing" del programma ESPRIT, è sviluppato da un consorzio europeo che vede Systems and Management come principale contraente e coinvolge:

- Bense Computer System (Germania Ovest)
- C.R.I.S.S. (Francia)
- Università di Dortmund (Germania Ovest)
- Università di Lione (Francia)
- Università di Pisa
- Enidata e Sipe Optimation (subcontraenti).

La durata prevista per il progetto è di 5 anni, con un costo totale di 7.000.000 di ECU (approssimativamente 10.000.000.000 di lire) ed un impegno complessivo di 73 anni/ uomo.

Obiettivo del progetto è studiare come la tecnologia delle basi di dati puo' essere impiegata per migliorare le prestazioni di una macchina logica e come d'altra parte un DBMS incorporato in un KBMS puo' trarre beneficio utilizzando la capacità di inferenza di una macchina logica. Il componente inferenziale fornisce la capacità di gestire regole e in generale manipolare conoscenza deduttiva, il componente data base fornisce una "collaudata" gestione di grosse quantità di dati.

Partendo dalle analogie esistenti tra la teoria dei data base relazionali e la deduzione basata sulla logica, vengono approfonditi alcuni aspetti che sono alla base della definizione di un KBMS in termini dell'integrazione fra la tecnologia Data Base e la tecnologia della programmazione logica.

Le attività di ricerca del progetto possono essere suddivise in tre linee principali.

(•) Il presente lavoro e' stato svolto per conto della Systems and Management SpA.

a) Integrazione DBMS - macchina di inferenza

L'approccio base di conoscenza puo' essere visto come una generalizzazione dell'approccio base di dati, potenziato dalla capacità di deduzione, e reso piu' uniforme dal momento che lo stesso paradigma viene utilizzato per definire i dati e le operazioni sia dipendenti che indipendenti dalla applicazione.

Dal punto di vista del DBMS, uno strato esterno basato sulla tecnologia dell'intelligenza artificiale, oltre alla capacità deduttiva, fornisce la possibilità di:

- affrontare sistematicamente problemi di verifica semantica, in particolare dei vincoli di integrità;
- estendere l'insieme delle funzionalità e introdurre i tipi degli oggetti, attraverso un ricco meccanismo di astrazione;
- usare un linguaggio dichiarativo per le applicazioni di gestione dei dati;
- definire interfacce utente "ricche di conoscenza";
- integrare basi di dati differenti attraverso un'unica interfaccia di interrogazione.

Dal punto di vista di un sistema basato sulla conoscenza (KBS), la tecnologia DB puo' essere vantaggiosamente utilizzata per gestire i fatti elementari. Cio' è molto importante per domini di conoscenza molto vasti dove la mancanza di struttura e l'inefficienza delle procedure di accesso ai dati renderebbero l'approccio KB non utilizzabile.

L'integrazione con un DBMS puo' fornire a un KBS anche altri benefici, come l'accesso concorrente alla conoscenza memorizzata nel DB, la protezione della stessa da accessi, modifiche o cancellazioni non autorizzati e le strategie di ripristino in caso di malfunzionamenti. L'integrazione delle due tecnologie permette inoltre di riutilizzare grossi DB già esistenti in nuove applicazioni basate sulla conoscenza.

In questa linea sarà anche studiata la rilevanza dei KBMS distribuiti per gestire KB molto grandi e la possibilità di utilizzare la tecnologia DB per gestire anche le regole.

b) Il linguaggio di rappresentazione della conoscenza

La maggior parte dei sistemi esperti della prima generazione mancando di un approccio sistematico, hanno causato la proliferazione di soluzioni dipendenti dalla specifica applicazione e quindi non riutilizzabili. La fattibilità di un approccio che miri a realizzare un sistema, le cui componenti siano in larga misura indipendenti dallo specifico dominio, si basa essenzialmente sulla scelta di un linguaggio "standard" per la rappresentazione della conoscenza. Il piu' generale e conosciuto tra i formalismi di rappresentazione della conoscenza, è la logica dei predicati del primo ordine. Inoltre il suo ricco sottoinsieme, la logica delle clausole di Horn (HCL) puo' essere gestito da un sistema di inferenza, ed essere utilizzato come linguaggio di programmazione con una intrinseca capacità deduttiva /Kowalski 74/. Seguendo questo approccio, il nucleo del linguaggio di rappresentazione della conoscenza scelto in Epsilon è l' HCL.

La scelta della programmazione logica come modello di rappresentazione della conoscenza non assume nessun significato di preclusione rispetto ad altri modelli quali le reti semantiche, i frames, i sistemi di produzione ed altri formalismi logici, che possono essere mappati in HCL e possono quindi essere utilizzati per ottenere differenti rappresentazioni esterne della conoscenza.

Tuttavia la sola logica delle clausole di Horn non puo' essere scelta come linguaggio di rappresentazione della conoscenza in un KBMS. Si rendono infatti necessarie estensioni linguistiche per fornire meccanismi di strutturazione e rendere piu' efficiente il processo di inferenza.

c) Strumenti per la definizione e l'utilizzazione di un KBMS

La terza linea di progetto è relativa alla definizione di strumenti per la definizione e l'utilizzo di un KBS. Un linguaggio logico e la relativa macchina di inferenza non sono infatti sufficienti per costruire applicazioni effettive in un KBS, ma, come nel caso della programmazione convenzionale, è necessario costruire un ambiente di programmazione /Ennals 84, Clark 82b, Hammond 82, Hammond 83, Hammond 84/, dotato di un insieme di strumenti, tra cui:

- strumenti per gestire l'interazione con l'utente, molto simili a quelli presenti negli ambienti integrati di programmazione avanzati, che forniscano interfacce orientate all'utente, strumenti di editing e di monitoring di sistema /Komorowski 82, Teitelbaum 81, Teitelman 79/;
- strumenti in grado di analizzare lo stato del processo di inferenza (quali sono le regole contenute nella base di conoscenza, quali deduzioni il sistema sta cercando di fare, perche il sistema chiede informazioni circa fatti specifici, ecc.) /Weir 82, Sergot 83/, necessari durante lo sviluppo e l'utilizzo dell'applicazione;
- strumenti di analisi semantica relativi alla verifica di proprietà /EPSILON 5/. Un esempio tipico è dato dalla verifica dei vincoli d'integrità in seguito all'inserzione o alla modifica di fatti.

L'obiettivo delle attività di ricerca sopra descritte consisterà nell'implementazione prototipale del nucleo di un KBMS in cui saranno considerati due tipi di integrazione: lasca (in cui la componente inferenziale si interfaccia al DBMS attraverso il linguaggio di interrogazione) e stretta (in cui si accede ai meccanismi interni di piu' basso livello del DBMS).

L'approccio della metaconoscenza e'alla base del progetto per sperimentare la realizzabilità di soluzioni uniformi ai diversi problemi evidenziati precedentemente. Nel prossimo paragrafo verranno esaminati la natura e il ruolo della metaconoscenza.

2. Metaprogrammazione e metaconoscenza

La metaconoscenza è un aspetto cruciale sia per quanto riguarda la rappresentazione della conoscenza sia per la risoluzione di problemi /Weyhrauch 80, Davis 80, Hayes-Roth 83, Aiello 84, Sterling 84a/. La metaconoscenza, "conoscenza sulla conoscenza", è definita da metaprogrammi che operano su domini come teorie, regole e prove. Poter rappresentare programmi come dati permette di implementare facilmente la metaconoscenza; nel caso dei linguaggi logici, le clausole sono rappresentate come termini (clausole = programmi - termini = dati) e quindi un linguaggio logico puo' essere usato sia come metalinguaggio che come linguaggio oggetto.

L'amalgamazione del linguaggio oggetto con il metalinguaggio /Bowen 82/ richiede la rappresentazione di tutte le entità sintattiche del linguaggio (oggetto) come strutture dati del metalinguaggio (termini), e la definizione dell'interprete del linguaggio oggetto (Demo) nel metalinguaggio (metainterprete). Il metainterprete è la rappresentazione a livello meta della relazione di provabilità del linguaggio oggetto. L'amalgamazione si basa su due principi di riflessione (regole di inferenza) /Weyhrauch 80, Bowen 82/, che, da una parte, preservano l'equivalenza fra le deduzioni effettuate nei due livelli e, dall'altra, permettono di riflettere una prova a livello meta in una prova (piu' efficiente) a livello oggetto. L'amalgamazione è una estensione conservativa, cioe' ogni cosa che puo' essere fatta nel linguaggio amalgamato puo' essere compiuta anche solo nel metalinguaggio (o nel linguaggio oggetto).

Inoltre:

a) L'amalgamazione è piu' espressiva del linguaggio oggetto, perche' permette di definire clausole che combinano relazioni a livello oggetto con relazioni a livello meta. Questa maggiore espressività è ottenuta in un modo puramente dichiarativo (senza funzionalità o primitive extra logiche) preservando la semantica standard della logica.

b) Il metainterprete puo' essere esteso con nuove funzionalità e nuove regole di inferenza. Cio' permette la definizione di nuove funzionalità del linguaggio (per esempio, la negazione e le teorie strutturate) con le corrispondenti regole di inferenza (la negazione come fallimento finito e l'ereditarietà multipla). La metaprogrammazione è gia parzialmente implementata in alcuni sistemi Prolog da un insieme appropriato di metapredicati primitivi, che permettono di definire sia il metainterprete Demo che altri metainterpreti per specifiche funzionalità.

Una delle caratteristiche principali della metaprogrammazione è la capacità di estendere il linguaggio, la macchina inferenziale e l'ambiente, senza dover modificare l'interprete Prolog. Cio' permette di riutilizzare altri strumenti, come per esempio un compilatore Prolog, che genera codice

molto veloce e potrebbe essere essenziale nel caso di applicazioni complesse. Strumenti definiti come metaprogrammi (Prolog) sono facilmente definibili, portabili (analogamente agli strumenti degli ambienti Lisp), e ottimizzabili tramite tecniche basate sulla valutazione parziale.

L'integrazione di conoscenza e metaconoscenza sembra essere un meccanismo di astrazione molto potente che permette di adattare il linguaggio e la macchina di inferenza alle necessità specifiche dell'utente, fornendo cosi' un'alternativa allo sviluppo di linguaggi di rappresentazione della conoscenza special-purpose e dei relativi strumenti.

L'alternativa alla metaprogrammazione è la definizione di effettive estensioni linguistiche (con le relative estensioni dell'interprete) e/o di specifici strumenti di analisi e verifica.

Nel paragrafo seguente si esaminerà come alcuni dei problemi relativi al controllo, alla rappresentazione della conoscenza e agli strumenti dell'ambiente logico possono essere risolti per mezzo della metaprogrammazione.

3. Controllo, rappresentazione della conoscenza e strumenti di gestione

a) *Controllo*

La metaconoscenza può essere utilizzata per controllare il processo di inferenza. Rendere efficiente il processo di inferenza è il problema chiave nella realizzazione di KBS. E' quindi importante essere capaci di sfruttare diversi tipi di metaconoscenza di controllo (cioe' conoscenza sul processo di inferenza). La metaconoscenza di controllo riguarda la definizione di una appropriata (dipendente dal problema) strategia di controllo dell'interprete: come selezionare il letterale che deve essere risolto e la regola che deve essere utilizzata per risolvere il letterale corrente, come controllare efficientemente il non determinismo e come ottimizzare il processo di ricerca. Alcuni sistemi Prolog permettono, seppure limitatamente, di definire la strategia di controllo fornendo degli specifici costrutti di controllo e/o delle annotazioni sulle variabili (in modo simile ai linguaggi logici concorrenti) /Clark81, Bellia82, Clark83, Shapiro83c/. Alcuni linguaggi logici /Hansson 82, Clark 82a, Naish 82, Barbuti 84, Barbuti 85/ permettono di definire regole di computazioni dinamiche simili alle coroutine per mezzo di annotazioni sulle variabili. Esistono inoltre dei sistemi Prolog (per esempio META-LOG /Dincbas84/), che forniscono un ricco insieme di metapredicati di controllo primitivi (indipendenti dal dominio). L'approccio seguito nel progetto Epsilon per la specifica a metalivello del controllo è descritta in /EPSILON 3/.

b) *Rappresentazione della conoscenza*

Questo aspetto si riferisce al problema di migliorare la capacità espressiva di HCL come linguaggio di rappresentazione della conoscenza. Le principali carenze di HCL dal punto di vista della rappresentazione della conoscenza sono:

- incapacità di gestire informazione negativa, problema ancora piu' grave se il programma logico viene interfacciato a un data base;
- mancanza di meccanismi di strutturazione della conoscenza;
- incapacità di gestire modelli di ragionamento non classici, quali il ragionamento non monotono o il ragionamento "per default".

E' già stato dimostrato che alcuni dei precedenti problemi possono essere risolti combinando conoscenza e metaconoscenza. Si veda, per esempio, /Kowalski 79b/ e /Bowen 82/, in cui sono introdotte soluzioni eleganti per alcuni di essi (ragionamento non monotono , frame problem, ecc).

La soluzione generale consiste dei seguenti passi:

- definizione di una estensione al linguaggio ;
- definizione di un metainterprete che incorpora le corrispondenti regole di inferenza. Per esempio, se la negazione è aggiunta al linguaggio, il metainterprete dovrebbe contenere una implementazione corretta della risoluzione SLDNF (risoluzione SLD + la negazione come fallimento finito).

I meccanismi di strutturazione della conoscenza devono includere costrutti statici (tipi e moduli) simili a quelli che sono stati proposti per i linguaggi di specifica e i linguaggi imperativi ad alto livello. Se la conoscenza sulle estensioni linguistiche è contenuta in metaprogrammi separati (vedi par. successivo sui tipi), questa non influenza l'interprete del livello oggetto e può essere utilizzata da un metainterprete che è essenzialmente uno strumento di analisi (e di verifica) statica.

Un altro meccanismo di strutturazione della conoscenza permette di definire una base di conoscenza come una collezione di corpi separati di conoscenza (sotto teorie separate non necessariamente consistenti), che possono essere correlati da una struttura gerarchica /Furukawa 84/. Se la struttura riflette un meccanismo di ereditarietà, questa regola di inferenza è implementata da un metainterprete adeguato, che "riconosce" la metaconoscenza riguardante i legami fra le teorie.

Una base di dati strutturata può anche essere vista come un insieme di componenti definiti utilizzando diverse tecnologie di rappresentazione della conoscenza. In questo caso la metaconoscenza è "ciò che ogni componente conosce dell'altro componente". In altre parole, la visione esterna (interfaccia) di una componente di conoscenza è rappresentata come metaconoscenza (nel caso di componenti basi di dati, l'interfaccia è qualcosa di simile a un dizionario dei dati).

Ultimo esempio di estensione linguistica, che può essere implementata a livello meta, è la definizione di un sottoinsieme di HCL che si comporti come un linguaggio funzionale. La motivazione principale di una tale estensione è la necessità di fornire un modo efficace per rappresentare la conoscenza procedurale nello stesso contesto di rappresentazione della conoscenza. Questo si può ottenere integrando Prolog e Lisp /Robinson 82a, Robinson 82b,Komorowski 82/, integrando HCL e le teorie equazionali /Gougen 84, Barbuti 84, Barbuti 85/ o definendo un sottoinsieme funzionale della logica delle clausole di Horn /Bellia 82, Bellia 84/. Le ultime due soluzioni sono piu' uniformi. La prima richiede soltanto un preprocessor che traduca equazioni in clausole /Barbuti 85/, mentre la seconda può essere ottenuta a livello meta forzando alcune variabili ad essere "read-only" e imponendo alcune restrizioni sintattiche.

c) *Strumenti dell'ambiente logico*

La metaprogrammazione può essere utilizzata per definire esplicitamente la conoscenza coinvolta nell'interazione utente-sistema, sia quando la base di conoscenza viene creata (per guidare l'acquisizione della conoscenza, la manutenzione, l'aggiornamento, l'analisi on-line, il monitoraggio interattivo dell'inferenza, ecc) sia quando viene utilizzata (spiegazione del ragionamento).

La prima classe di strumenti, definiti come metaprogrammi, permette di definire in modo completamente dichiarativo le operazioni di gestione della base di conoscenza (assert, retract, ecc.), che sono necessarie in ogni sistema di programmazione logica.

Con lo stesso approccio si può definire la verifica di proprietà della base di conoscenza. Un aspetto importante di ogni sistema di gestione di basi di conoscenza avanzato è, infatti, la possibilità di asserire proprietà che devono essere soddisfatte dalla base di conoscenza e di verificarle in caso di modifica della stessa . Un ben noto esempio riguardante le basi di dati è la verifica dei vincoli di integrità. Le proprietà della KB possono essere asserite in termini di metaregole che saranno utilizzate ogni volta che la base di conoscenza verrà modificata (cioe' quando si compie una operazione di aggiornamento). La metaregola asserisce la condizione che deve essere verificata. In caso di aggiornamento di fatti elementari, la condizione potrebbe essere semplicemente un'interrogazione che può essere verificata dall'interprete; in caso di modifica delle regole, le condizioni da verificare richiedono in generale la piena capacità del "theorem-proving".

Un'altra area per gli strumenti di verifica è quella della metaconoscenza orientata alla strutturazione o, piu' in generale, all'estensioni linguistiche "statiche" (tipi, interfacce fra i moduli, vincoli sintattici al sottoinsieme funzionale, ecc).

Le operazioni di gestione della KB possono essere estese da appropriate definizioni al meta livello in un'altra direzione, cioe' il "forward reasoning", che non è in genere disponibile nei sistemi di programmazione logica ed è molto utile in situazioni tipiche dei sistemi basati sulla conoscenza (simulazione, analisi what-if, propagazione dei vincoli, ecc).

La seconda classe di strumenti contiene metainterpreti che forniscono facilitazioni tipiche dei sistemi esperti (spiegazione, query-the-user,ecc.) /Sergot 83, Walker 83, Bowen 84, Sterling 84b/, o capacità di monitoring interattivo (debugger, tracer, ecc.) /Shapiro 83b/.

4. Tipi e moduli in Epsilon

I risultati della ricerca nel campo del "software engineering" hanno dimostrato che la modularizzazione e la definizione di tipi di dati flessibili, sono i meccanismi di strutturazione necessari nella produzione di software su larga scala; analogamente nel campo dell'Intelligenza Artificiale è sorta la necessità di meccanismi di strutturazione per rappresentare la conoscenza. I due approcci, pur riflettendo i diversi interessi delle due aree (strutturazione dei programmi e strutturazione della conoscenza), possono essere combinati per ottenere un linguaggio piu' potente e flessibile.

Nel primo anno di lavoro del progetto Epsilon, è stata studiata un'estensione del Prolog con meccanismi di strutturazione (**tipi, moduli e teorie**), utilizzando l'approccio della metaprogrammazione.

Alcune strutture di tipi sono state proposte per estendere il Prolog /Bruynooghe 82, Mycroft 83/. In /Asirelli 83/ è definita sia la semantica model theoretic del linguaggio esteso, sia una semantica operazionale, che consiste nel controllo statico dei tipi piu' la semantica operazionale standard di HCL.

In Epsilon, come primo passo verso un linguaggio di programmazione logica strutturato, è stato esteso il Prolog con un **sistema di tipi** contenente sia la definizione sintattica, sia il controllo di correttezza.

L'algoritmo di checking (realizzato utilizzando la semantica operazionale definita in /Asirelli 83/) garantisce la correttezza dei tipi di un programma eseguendolo in un dominio non-standard: la teoria dei tipi derivata dal programma stesso.Il type-checker è un metainterprete che legge il programma, genera la teoria che descrive la struttura dei tipi e lo esegue in quella teoria; quest'ultimo passo può essere eseguito dall'interprete standard del linguaggio logico, chiamato dal metainterprete.

Una struttura di tipi equivalente a quella del linguaggio proposto può essere aggiunta a programmi Prolog standard definendo i tipi come metapredicati, questo permette di scrivere programmi tipati interamente in Prolog. In entrambi i casi si utilizza la stessa procedura di checking.

Il type-cheker offre due modalità di controllo del programma:

- sottoporre l'intero programma allo strumento;
- introdurre incrementalmente una clausola alla volta.

I **moduli** possono essere aggiunti al linguaggio definito in Epsilon, seguendo due approcci differenti:

- come costrutto di strutturazione del programma /Mprolog 84, Furukawa 83,Gougen 84/ ;
- come costrutto di strutturazione della conoscenza (**teorie**) /Furukawa 84/.

La metodologia per gestire i moduli nel primo approccio è facilmente derivata da quella utilizzata per i tipi, estendendo il linguaggio Prolog con una struttura di moduli e controllando le loro interfacce per mezzo di un metanterprete che genera la teoria della struttura dei moduli ed esegue il programma in quella teoria.

Nel secondo approccio è necessario definire sia un meccanismo generale che permetta la gestione delle teorie che un insieme di metapredicati specifici che definiscano le relazioni tra esse. Questo permette l'implementazione di metainterpreti che lavorano su universi (la composizione di teorie e relazioni tra esse).

La base di conoscenza di Epsilon può essere suddivisa in moduli di conoscenza monotona (teorie) identificate da un nome. L'informazione sui legami fra le teorie non è inclusa nelle teorie stesse, ma è contenuta in una specifica metateoria, la gestione dell'insieme delle teorie è quindi assicurato dal controllo a metalivello.

Strutturare la conoscenza in teorie permette di simulare in modo naturale il ragionamento per default e quello non monotono. Teorie separate possono anche essere utili in relazione ai meccanismi di ragionamento "in avanti" e per il ragionamento ipotetico.

Anche se i possibili legami fra le teorie non sono stati ancora definiti completamente, è possibile definirne alcuni basici e crearne altri dinamicamente. Il primo legame che si può definire fra due teorie è quello gerarchico: T1 is-a-subtheory-of T2, in cui la teoria T1 è una specializzazione della teoria T2; un'altro tipo di legame può essere definito fra una teoria e una sua metateoria: T1 is-a-metatheory-of T2, in cui T1 include informazioni sulle proprietà della teoria T2, per esempio i tipi, i vincoli di integrità, informazioni sul controllo, ecc. Una metateoria T1 può gestire una teoria

T2: T1 is-a-manager-of T2, se T1 contiene informazioni su come gestire l'aggiornamento della teoria T2. Le teorie possono essere legate anche in modo non gerarchico, e in questo caso è necessario definire le regole di visibilità che il legame esprime. Per esempio, si può definire un legame T1 is-a-filter-of T2, in cui la metaconoscenza contenuta in T2 è ciò che T1 conosce di T2, e T1 si comporta come una interfaccia di T2 per le altre teorie. Questo tipo di legame può essere utile per gestire l'interfaccia fra il linguaggio logico e una base di dati relazionale: T1 è in questo caso il dizionario della base di dati espresso in Prolog e viene utilizzato per tradurre interrogazioni logiche in interrogazioni alla base di dati.

Anche nel caso di teorie separate è necessario stabilire se la specifica metaconoscenza può essere gestita piu' adeguatamente, dal punto di vista delle prestazioni, da estensioni del linguaggio (e quindi dell'interprete) o dall'amalgamazione di conoscenza e metaconoscenza utilizzando un metainterprete. Cioe' le teorie possono essere simulate in un Prolog standard o esser implementate estendendo il linguaggio con metapredicati primitivi di gestione delle teorie.

Riferimenti Bibliografici

/Aiello 84/ **L.Aiello and G. Levi**, The uses of metaknowledge in AI systems. Proc. ECAI84, Advances in Artificial Intelligence, T. O'Shea Ed. (Elsevier Science Publishers, 1984), 705-717.

/Asirelli 83/ **P.Asirelli, R.Barbuti, G.Levi**, Types and declarative static type checking in logic programming, to be published in "Programming '84".

/Barbuti 84/ **R. Barbuti, M. Bellia, G. Levi** and **M. Martelli**, On the integration of logic programming and functional programming. Proc. 1984 Int. Symp. on Logic Programming (IEEE Comp. Society Press, 1984), 160-166.

/Barbuti 85/ **R. Barbuti, M. Bellia, G. Levi** and **M. Martelli**, LEAF: A language which integrates logic, equations and functions. In Logic Programming: Functions, Relations and Equations, D. DeGroot and G. Lindstrom, Eds. (Prentice-Hall, 1985).

/Bellia 82/ **M. Bellia, E. Dameri, P. Degano, G. Levi** and **M. Martelli**, Applicative communicating processes in first order logic. Proc. 5th Int. Symp. on Programming, LNCS 137 (Springer Verlag, 1982), 1-14.

/Bellia 84/ **M. Bellia, E. Dameri, P. Degano, G. Levi** and **M. Martelli**, A formal model for lazy implementation of a PROLOG compatible functional language. In Implementations of PROLOG , J.A. Campbell, Ed. (Ellis Horwood, 1984), 309-326.

/Bowen 82/ **K.A. Bowen** and **R.A. Kowalski**, Amalgamating language and metalanguage in logic programming. In Logic Programming, K.L. Clark and S.-A. Tarnlund, Eds. (Academic Press, 1982), 153-172.

/Bowen 84/ **K.A. Bowen**, Expert systems programming in metaPROLOG, unpublished manuscript (1984).

/Bruynooghe 82/ **M.Bruynooghe**, Adding redundancy to obtain more reliable and readable Prolog programs, Proc. First Int. Logic Programming Conf., Marseille, Sept. 1982.

/Clark 81/ **K.L. Clark** and **S. Gregory**, A relational language for parallel programming. Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (1981), 171-178.

/Clark 82a/ **K.L. Clark, F.G. McCabe and S. Gregory**, IC-Prolog Language Features, in Logic Programming, Clark K. and Tarnlund S.-A, eds. (Academic Press, 1982).

/Clark 82b/ **K.L. Clark and F.G. McCabe**, PROLOG: a language for implementing expert systems. Machine Intelligence 10 (1982), 455-470.

/Clark 83/ **K.L. Clark** and **S. Gregory**, PARLOG: a parallel logic programming language. Imperial College Research Report 83/5 (May 1983).

/Davis 80/ **R. Davis**, Meta-rules: Reasoning about control. Artificial Intelligence 15 (1980), 179-182.

/Dincbas 84/ **M. Dincbas and J.P. Lepape**, Metacontrol of logic programs in METALOG, Proc. Int'l Conf. on Fifth Generation Computer Systems (1984), 361-370.

/Ennals 84/ **R. Ennals, J. Briggs and D. Brough**, What the Naive User Wants from Prolog, in Implementations of Prolog, J.A Campbell, ed. (Ellis Horwood, 1984), 376-386.

/EPSILON 3/ Control of Logic Programs: a survey and a proposal. Epsilon Internal Report, November 1985.

/EPSILON 5/ An Introduction to the proof of logic program properties. Epsilon Internal Report, November 1985.

/Furukawa 83/ **K.Furukawa, R.Nakajima, and A.Yonezawa**, Modularization and abstraction in logic programming, New Generation Computing 1, (1983).

/Furukawa 84/ **K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki and K. Ueda**, Mandala: A logic based knowledge programming system, Proc. Int'l Conf. on Fifth Generation Computer Systems (1984), 613-622.

/Goguen 84/ **J.A. Goguen and J. Meseguer**, Equality, types, modules and (why not?) generics for logic programming. J. Logic Programming 1 (1984), 179-210.

/Hammond 82/ **P. Hammond**, Appendix to Prolog: A language for implementing expert systems. Machine Intelligence 10 (1982).

/Hammond 83/ **P. Hammond and M. Sergot**, A PROLOG shell for logic based expert systems. Techn. Report, Imperial College (1983).

/Hammond 84/ **P. Hammond**, Micro-PROLOG for expert systems. In Micro-PROLOG: Programming in Logic (Prentice Hall, 1984).

/Hayes-Roth 83/ **F. Hayes-Roth, D. Waterman and D. Lenat**, Building expert systems (Addison-Wesley, 1983).

/Hansson 82/ **A. Hansson, S. Haridi and S.-A. Tarnlund**, Properties of a logic programming language. In Logic Programming, K.L. Clark and S.-A. Tarnlund, Eds. (Academic Press, 1982), 267-280.

/Komorowski 82/ **H.J. Komorowski**, QLOG - The Programming Environment for Prolog in Lisp, in Logic Programming, Clark K. and Tarnlund S.-A, eds. (Academic Press, 1982).

/Kowalski 74/ **R.A. Kowalski**, Predicate Logic as a Programming Language, Proc.IFIP74 (North-Holland, 1974).

/Kowalski 79b/ **R.A. Kowalski**, Logic for problem solving (North Holland, 1979).

/Mprolog 84/ Mprolog Language Reference Manual Release 2.1, December 1984.

/Mycroft 83/ **A.Mycroft and R.O'Keefe**, A polymorphic type system for Prolog, Proc. Logic Programming Workshop'83, Algarve, Portugal.

/Naish 82/ **L. Naish**, An introduction to MU-Prolog, Dept. of Comp. Sc., University of Melbourne (1982).

/Robinson 82a/ **J.A. Robinson and E.E. Sibert,** LOGLISP: Motivations, design and implementation. In Logic Programming , K.L. Clark and S.-A. Tarnlund, Eds. (Academic Press, 1982), 299-314.

/Robinson 82b/ **J.A. Robinson and E.E. Sibert,** LOGLISP: An alternative to PROLOG. Machine Intelligence 10 (Ellis Horwood, 1982).

/Sergot 83/ **M. Sergot**, A query-the-user facility for logic programs, in Integrated Interactive Computer Systems, P. Degano and E. Sandewall, Eds. (North Holland, 1983).

/Shapiro 83b/ **E.Y. Shapiro**, Algorithmic program debugging (MIT Press, 1983).

/Shapiro 83c/ **E.Y. Shapiro**, A subset of Concurrent Prolog and its interpreter. Techn. Rep. TR-003, ICOT (1983).

/Sterling 84a/ **L.S. Sterling**, Logical levels of problem solving. J. of Logic Programming 2 (1984).

/Sterling 84b/ **L.S. Sterling**, Expert system = knowledge + meta-interpreter, Technical Report (1984).

/Teitelbaum 81/ **R. Teitelbaum and T. Reps**, The Cornell Program Synthesizer: A syntax-directed programming environment. Comm. ACM 24 (1981).

/Teitelman 79/ **W. Teitelman**, INTERLISP Reference Manual, Xerox PARC (1979).

/Walker 83/ **A. Walker**, Prolog/EX1, an inference engine which explains both yes and no answers. Proc. IJCAI 8 (1983), 526-528.

/Weir 82/ **D. Weir**, Interactive Facilities for micro-Prolog, M.Sc. Dissertation. Dept. of Computing, Imperial College, 1982.

/Weyhrauch 80/ **R. Weyhrauch**, Prolegomena to a Theory of Mechanized formal Reasoning, Artificial Intelligence 13 (1980), 133-170.

LOGIFORM: UNO SPREADSHEET DEDUTTIVO

A.Bonsignori(*), N.Ciaramella(*), G.San Martini(*), F.Turini(**)

(*) Sipe Optimation Laboratorio Ricerca e Sviluppo
(**) Università di Pisa Dipartimento di Informatica

ABSTRACT

Logiform è uno strumento per la costruzione, la rappresentazione e l'utilizzo di una base di conoscenza. Una interfaccia di tipo spreadsheet facilita l'acquisizione della conoscenza e la sua modifica e utilizzazione. Lo strumento aiuta l'utente nel riempimento dello spreadsheet, mostrando le conseguenze logiche delle operazioni di inserzione e rimozione della conoscenza e suggerendo ulteriori operazioni. Logiform è implementato in Prolog, con ampio uso della programmazione a livello meta che consente di realizzare semplicemente modalità non standard di interpretazione.

1.Introduzione

L'idea guida del progetto Logiform è di costruire un sistema basato su un paradigma semplice ed efficace di interazione con un knowledge base che utilizzi la logica [Gen] (in pratica il Prolog) come linguaggio di rappresentazione e manipolazione della conoscenza, sfruttandone al meglio le caratteristiche di naturalezza ed espressività.

Per raggiungere tali obiettivi ci si è concentrati su una classe di problemi, peraltro di grande diffusione e rilevanza pratica: i problemi che hanno struttura di reti di vincoli [Pan]. Per tali problemi è adeguata una modalità di interazione a spreadsheet, ormai affermatasi come la più facilmente accessibile ad un utente non tecnico.
Lo strumento Logiform consente all'utente di esprimere le sue richieste ad un knowledge base (KB) in termini di caselle dello spreadsheet, che sono a loro volta immagini della conoscenza contenuta nel KB stesso. Nel suo lavoro l'utente collabora col sistema, che può avere a disposizione altre sorgenti di conoscenza che gli permettono di manipolare il KB; in questa ottica l'utente stesso è una fra le sorgenti di conoscenza.
Tali sorgenti di conoscenza sono utilizzate dal sistema per risolvere i problemi che si presentano, in base ad una descrizione delle loro caratteristiche; in particolare il sistema possiede una descrizione di aspetti strutturali e semantici del KB.
Queste descrizioni guidano l'esecuzione dei programmi logici che costituiscono Logiform, consentendo di realizzare modalità di interpretazione non standard: per questo scopo è sistematicamente utilizzata la programmazione a livello meta, ed è questo l'aspetto implementativo più interessante di Logiform.

La funzionalità più appariscente di Logiform, che è una estensione di quelle degli spreadsheet

convenzionali, è quella di propagazione della conoscenza. Quando l'utente inserisce informazione riempiendo caselle dello spreadsheet il sistema ne trae le conseguenze logiche con meccanismi di tipo forward reasoning e le visualizza, riempiendo così ulteriori caselle. Inoltre Logiform può aiutare l'utente nel riempimento di caselle suggerendogli dei valori ammissibili.

Particolare attenzione è rivolta all'acquisizione e utilizzo di conoscenza incompleta, approssimata o generica: mediante esecuzione parziale dei programmi logici i problemi che non possono al momento essere risolti vengono ridotti in forma più semplice sfruttando la conoscenza disponibile. Questa tecnica di esecuzione, analogamente all'interpretazione simbolica nel senso tradizionale, consente di circoscrivere le soluzioni dei problemi, o anche di esprimerle in forma più compatta e significativa.
L'esecuzione parziale è altamente desiderabile in un ambiente di interattività spinta quale Logiform, dove la conoscenza cresce incrementalmente attraverso l'interazione con altre sorgenti (primo fra tutti l'utente); per la sua realizzazione si ricorre ancora all'interpretazione a livello meta dei programmi logici.

Come in genere i sistemi esperti [Coh,Dude,Ham2,Clark,Hay] (e Logiform ha alcune caratteristiche di una shell per sistemi esperti), il sistema ha la capacità di spiegare il proprio comportamento: in effetti Logiform possiede anche una descrizione del proprio funzionamento e può quindi descrivere le sue stesse operazioni [Cec].

2. Applicazioni e metodologia

La costruzione della base di conoscenza di un'applicazione Logiform e' guidata dall'idea di specificare il problema sotto forma di rete di vincoli. D'altra parte, la classe dei problemi che presentano una struttura profonda a reti di vincoli è sufficientemente ampia e, come vedremo, ben si adatta alle caratteristiche della programmazione logica ed a quelle della tecnologia degli spreadsheet.

Il problema della soluzione di reti di vincoli può essere enunciato in una forma molto generale come segue:

> *Dato un insieme di variabili, dette nodi, e un insieme di relazioni tra variabili, dette vincoli, assegnare un valore ad ogni variabile in modo da soddisfare ogni vincolo.*

In questa formulazione così generale rientrano moltissimi problemi di grande interesse sia teorico che pratico; in particolare questo modello risulta molto naturale per problemi in cui si devono effettuare una serie di scelte fra molte soluzioni possibili, rispettando certe regole di compatibilità fra di esse. Problemi di tal genere sono quelli di scheduling, pianificazione di attività, distribuzione geografica di servizi, progettazione di sistemi tramite composizione di sottosistemi, riconoscimento di immagini; in generale possiamo parlare di problemi di "progettazione in presenza di vincoli" e di "riconoscimento di configurazioni".

Una rete di vincoli può essere l'insieme dei vincoli di integrità di un data base, e in questo caso la rete ha per soluzione ogni stato globale dei dati (i nodi della rete) consistente con tali vincoli di integrità: in questo esempio la rete di vincoli ha un significato puramente analitico, nel senso che i valori dei nodi provengono da una fonte esterna. La rete di vincoli può anche svolgere una funzione sintetica e costruttiva, come in sistema di equazioni, viste come vincoli sulle incognite che costituiscono i nodi.

In un ambiente interattivo ed incrementale quale Logiform è di particolare interesse la combinazione di questi due aspetti di una rete di vincoli, in cui i valori esogeni vengono controllati per verificarne la compatibilità con i vincoli, e poi utilizzati per generare altri valori e avvicinarsi ulteriormente alla soluzione completa della rete.

Dagli esempi si può comprendere come un'interfaccia spreadsheet risulti essere del tutto naturale per molti problemi espressi come reti di vincoli: basti pensare alla compilazione di un budget o di un orario, in cui le caselle dello spreadsheet rappresentano gli oggetti e le annotazioni esprimono i vincoli.
Quando l'utente riempie delle caselle e lo spreadsheet risponde riempiendone a sua volta altre, i due stanno collaborando alla soluzione di una rete di vincoli.

La programmazione logica è particolarmente indicata per esprimere problemi modellati come reti di vincoli, sia per le caratteristiche di dichiaratività e interattività, sia perché orientata più al concetto logico-matematico di relazione, di impiego naturale nei problemi citati, che non a quello di funzione su cui si basa la programmazione convenzionale.

3.Interfacciamento con la base di conoscenza

La principale modalità con cui l'utente di Logiform accede al KB è attraverso uno spreadsheet, sia in visualizzazione che in interrogazione e modifica; in questo modo si ottiene una notevole immediatezza e semplicità di interazione.

Per illustrare le modalità di questa interazione ci riferiamo ad un semplice esempio: un KB che descrive l'orario delle lezioni in una scuola.

Nel KB sono presenti fatti come "il professor Rossi tiene lezione di matematica il giovedì alle 11 nell'aula 8". Per rappresentare fatti di questo genere il formato tabellare è chiaramente il più naturale, e in effetti è impiegato spontaneamente dai compilatori di orari. L'utente di Logiform può creare una rappresentazione (parziale) dell'orario contenuto nel KB mediante una istruzione come "visualizza le lezioni fissando l'aula 8, elencando i giorni in orizzontale a partire dalla casella B1, le ore in verticale a partire dalla casella A2, e i relativi corsi nelle caselle di intersezione".
In risposta a tale richiesta Logiform crea un mapping tra le caselle dello spreadsheet e i fatti del KB, utilizzando una descrizione delle proprietà di quest'ultimo (ad esempio il fatto che in una certa aula non si possono svolgere simultaneamente due lezioni). Riempie quindi lo spreadsheet in base al contenuto del KB: in particolare inserirà nella casella E1 il valore 'giovedì', nella casella A4 il valore '11' (supponendo che il primo giorno sia lunedì e la prima ora le 9), e nella casella di incrocio E4 il valore 'matematica'.
L'utente può adesso posizionarsi sulla casella E4 e chiedere "professore?" ottenendo in risposta 'Rossi', oppure modificare 'matematica' in 'fisica', e in tal caso il sistema elimina dal KB la lezione esistente, sostituendola con "un professore ignoto tiene lezione di fisica il giovedì alle 11 nell'aula 8".

L'estrema elasticità espressiva permessa dalla programmazione logica consente di mantenere diverse viste personalizzate di uno stesso KB in modo semplice: il professor Rossi può chiedere "visualizza le lezioni fissando il professor Rossi, elencando i giorni in orizzontale a partire dalla casella X, le ore in verticale a partire dalla casella Y, e i relativi corsi ed aule nelle caselle di intersezione", visualizzando lo stesso KB in modo diverso.

4.Meccanismi di propagazione

La conoscenza introdotta dall'utente attraverso lo spreadsheet viene controllata dal sistema che ne verifica la consistenza rispetto al KB [Bow1]. Essa viene poi utilizzata per inferire nuova conoscenza con meccanismi data-driven. La conoscenza così acquisita è poi utilizzata per riempire caselle sullo spreadsheet, oppure per aiutare l'utente nel riempimento.
Il meccanismo di propagazione si basa sulla descrizione in Prolog degli oggetti che rappresentano i nodi della rete di vincoli; i vincoli stessi sono espressi come regole Prolog.
Logiform applica queste regole alle descrizioni degli oggetti e ne ricava nuova conoscenza sugli oggetti stessi con cui arricchisce le loro descrizioni. E' particolarmente utile in questo contesto la possibilità offerta dalla programmazione meta di manipolare programmi logici come dati.
Tornando all'esempio dell'orario scolastico supponiamo che esista una regola che impone che le lezioni di educazione fisica si svolgano alle 12, che nello stato attuale tutte queste ore siano già impegnate tranne quelle del mercoledì e del venerdì e che debba ancora essere assegnata ancora una lezione di educazione fisica. In tal caso, se l'utente assegna una lezione di matematica alle 12 del mercoledì, il meccanismo di propagazione verifica che una soluzione legale del vincolo precedente può essere ottenuta solo assegnando la lezione di educazione fisica alle 12 del venerdì .
Questo comporta il riempimento di una casella dello spreadsheet e la modifica della descrizione dell'ora stessa, che adesso risulta occupata e quindi non disponibile per altre lezioni; l'applicazione di altre regole prosegue con la descrizione così modificata.
Logiform converte la descrizione degli oggetti e delle regole della rete di vincoli in una rappresentazione interna che si presenta sintatticamente come un insieme di goal Prolog (detti annotazioni):

$$\begin{array}{l} <- p_{11},\ldots,p_{1k(1)} \\ \quad . \qquad\qquad . \\ \quad . \qquad\qquad . \\ <- p_{n1},\ldots,p_{nk(n)} \end{array}$$

dove i p_{ij} sono letterali Prolog.

Gli oggetti sono descritti attraverso i valori (noti o incogniti) delle loro proprietà; questi valori sono rappresentati da variabili, che devono quindi essere condivise dalle annotazioni del sistema. La condivisione delle variabili, non supportata direttamente dal Prolog, è gestita da un metainterprete.
I vincoli della rete sono quindi rappresentati dalle annotazioni: una soluzione della rete è un assegnamento di valori alle variabili (condivise e non) che soddisfa simultaneamente tutte le annotazioni, viste come goal Prolog.
L'evoluzione della rete avviene mediante assegnamenti di valori alle proprietà degli oggetti, rappresentati da assegnamenti di valori alle variabili condivise del sistema di annotazioni.
Questo assegnamento può essere provocato dal riempimento di caselle dello spreadsheet da parte dell'utente, oppure da trasformazioni che Logiform stesso effettua per approssimare una soluzione.
Una trasformazione consiste nel determinare un assegnamento di variabili che soddisfa un p_{ij}, inteso come goal Prolog. Tale assegnamento è propagato a tutto il sistema, generandone un altro, più istanziato del precedente, che ha come soluzioni un sottoinsieme di quelle originali.
Quindi una trasformazione è una esecuzione parziale della descrizione dello stato corrente di una rete di vincoli, che genera una descrizione dello stato risultante dall'acquisizione di conoscenza dall'utente e dall'inferenza delle conseguenze logiche di questa. In sintesi, questo comportamento e' dovuto ad un meccanismo che reagisce alla modifica dei fatti applicando le regole opportune e generando nuovi fatti, il che, nella sostanza, equivale ad un uso di tipo forward reasoning della

conoscenza.

5.Esecuzione parziale

Continuando a riferirci all'esempio dell'orario scolastico, poniamo che il KB contenga questa conoscenza:

- le lezioni di matematica si svolgono in aule con almeno 50 posti;
- l'aula 1 ha più di 50 posti;
- il martedì alle 10 l'aula 1 è occupata da una lezione già assegnata;
- in un'aula non si possono svolgere più lezioni contemporaneamente.

In tal caso ad una casella dello spreadsheet che, per le regole di configurazione dello spreadsheet stesso, deve contenere l'aula in cui si svolge la lezione di matematica del martedì alle 10 contiene il valore (inserito dal sistema) 'un'aula con più di 50 posti, ma non l'aula 1'. Un tale comportamento può essere ottenuto da un interprete non standard che non risolve sino in fondo il problema di trovare un'aula adatta, se non richiesto esplicitamente, ma riduce il problema stesso ad uno più circoscritto, facilitando così all'utente il compito di dare una risposta definita.
In termini di programmazione logica ciò significa che un goal fallisce se si verifica una vera e propria mancanza di soluzioni ammissibili, e non per insufficiente informazione. Tale comportamento può essere, ancora una volta, convenientemente realizzato con una interpretazione a livello meta.

6. Spiegazione dell'inferenza

Con un metainterprete si può realizzare semplicemente la funzionalità di spiegazione delle inferenze: ad una deduzione il metainterprete associa una traccia delle regole e dei fatti utilizzati; la ricostruzione avviene poi secondo modalità fissate dall'utente stesso.
Per i particolari rimandiamo a [Cec].

7.Schema di una sessione

Per comprendere meglio quanto detto è utile descrivere le fasi salienti di una sessione di lavoro con Logiform:

- L'utente (o il programmatore applicativo per lui) introduce una descrizione del KB.
- L'utente configura lo spreadsheet per la visualizzazione del KB.
- L'utente riempie alcuni campi dello spreadsheet, a mano o indicando come ottenerli da un database connesso a Logiform.
- Logiform esamina i dati (fatti del KB), e scopre che essi violano i vincoli; segnala questo errore, e su richiesta giustifica il suo rifiuto mostrando la violazione del vincolo.

- L'utente decide se modificare i vincoli oppure i dati; in entrambi i casi effettua una nuova transazione.

- Logiform verifica la correttezza dell'operazione; scopre inoltre che i nuovi dati permettono di trovare una soluzione "buona" (algoritmicamente determinata o euristicamente giudicata soddisfacente) per qualche vincolo, e assegna gli opportuni valori alle variabili, riempiendo così alcune caselle dello spreadsheet.

- L'utente chiede il perché di questa azione di Logiform, e questi spiega sinteticamente le regole che gli hanno consentito questa inferenza. A questo punto l'utente può modificare dati o vincoli e anche tornare indietro, insoddisfatto dell'esito della transazione.

- L'utente è indeciso sul valore da assegnare ad una certa casella, e chiede un suggerimento a Logiform; questi mostra quali vincoli coinvolgono quella casella e quali valori è (attualmente!) possibile inserirvi. Questa risposta non viene data a priori, ma dipende dallo stato attuale dello spreadsheet: quanto più ci si avvicina alla soluzione, tanto più precisa è la risposta. In altri termini l'utente può vedere l'evoluzione non solo dei dati ma anche delle regole che governano il suo problema. Questo è reso possibile dal fatto che per Logiform le regole stesse sono dati manipolabili.

- L'utente è indeciso sulla strategia di riempimento delle caselle e chiede un suggerimento a Logiform.

- Logiform individua dei vincoli su cui è conveniente puntare l'attenzione, assume l'iniziativa del dialogo e chiede all'utente (o ad altre sorgenti di conoscenza) i valori per certe caselle da cui può trarre la soluzione del vincolo.

8. Conclusioni

Logiform è uno strumento che consente la definizione e l'utilizzo di basi di conoscenze per domini a "reti di vincoli", in cui la principale, ma non esclusiva, modalità di interazione è basata sullo spreadsheet come da noi inteso.

Per l'implementazione di Logiform si è scelto il Prolog [Kow3,Clock], attualmente la migliore approssimazione all'ideale della programmazione logica disponibile come linguaggio praticamente utilizzabile. La strategia di implementazione seguita si rifà all'approccio della programmazione a livello meta [Bow1, Bow2, Sterl], che consente di costruire con minimo sforzo più interpreti del linguaggio, utilizzabili in contesti e per scopi diversi. Con tale tecnica le varie funzionalità di Logiform sono implementate da moduli che utilizzano in modi diversi descrizioni (uniche) delle varie fonti di conoscenza disponibili.

Le caratteristiche presentate costituiscono il nucleo a partire dal quale sarà possibile costruire supporti più sofisticati, in particolare:

- la gestione della metaconoscenza.
  L'utilizzo dell'informazione di controllo e' cruciale per ottenere un'efficienza che sia accettabile per l'utente, esigenza tanto più sentita quanto più è complesso il problema [Aie]. Logiform

prevede di eseguire le proprie funzionalità tenendo conto sia di strategie dipendenti dal dominio che di informazioni di controllo più generali [Per] relative al funzionamento stesso dello strumento. Le prime potranno essere programmate in fase di generazione dell'applicazione, utilizzando ancora lo stesso linguaggio di rappresentazione della conoscenza; mentre per le altre esistera' una libreria di sistema. Infine faranno parte di quest'ultime anche quelle euristiche, per'altro generiche, che sotto certe ipotesi, permettono una riduzione della complessità di reti di vincoli.

- L'acquisizione di conoscenza.

  A tale proposito e' necessario gestire questa conoscenza in modo dinamico ed incrementale, considerando anche che e' espressa in funzione della visione (esterna) della conoscenza del particolare utente. La verifica della consistenza della annotazione introdotta dall'utente, o della sua trasformazione secondo il mapping, sarà eseguita come prova di proprietà (rappresentata dalla nuova annotazione), di un programma logico (il knowledge base) [Asi], basandosi anche su tecniche di esecuzione simbolica;

- Gestione contestuale dello stato.

  Affinche' l'utente possa formulare ipotesi ed eventualmente ritrattarle, e' necessario che Logiform gestisca un albero di contesti (o scenari), su cui muoversi a secondo della particolare richiesta dell'utente: la formulazione di un ipotesi o l'accettazione di un suggerimento, deve far si che il sistema passi in un nuovo contesto; naturalmente lo scenario precedente sara' riattivato in caso di annullamento dell'ipotesi o rifiuto del suggerimento.

Bibliografia

[Aie] Aiello, L. and Levi, G., "The use of metaknowledge in AI systems",*ECAI 84: Advances in Artificial Intelligence*, Ed. T O'Shea, 1984, pp 705-717.

[Asi] Asirelli, P.,Barbuti, R. and Levi, G.,"An introduction to the proof of logic program properties", Tech.Rep. ESPRIT Project 530 (1985).

[Bow1] Bowen, K.A. and Kowalski, R., "Amalgamating Language and Metalanguage in Logic Programming", in *Logic Programming*, (Clark and Tarnlund Eds), Academic Press, 1982.

[Bow2] Bowen, K.A. and Weinberg, T. "A meta-level extension of Prolog",Proc. 1985 Symp. on Logic Programming, IEEE Comp. Society Press, 1985.

[Cec] Cecchini, F., "Un modulo di spiegazione per sistemi esperti", Primo Conv. *GULP*, Genova, 1986.

[Clark] Clark, K.L., "Prolog a language for implementing Expert Systems",*Machine Intelligence* 10, (Hayes and Michie Eds).

[Clock] Clocksin, W.F. and Mellish, C.S.,"Programming in Prolog",Springer Verlag, New York, 1981.

[Coh] Cohen, P.R. and Feigenbaum, E.A. (Eds), "The Handbook of Artificial Intelligence",Vol II, W. Kaufmann, Los Altos,CA, 1981.

[Dude] Dude, R.O. and Gashing, J.G., "Knowledge based expert systems coming of age",*BYTE*, 6, 9 (Sept. 81), pp 238-278.

[Gen] Genesereth, M.R. and Ginsberg, M.,"Logic Programming", *Comm. ACM*, 28, 9 (Sept. 85) pp 933-941.

[Ham1] Hammond, P.,"Appendix to Prolog: a language for implementing expert systems", Technical Report, Imperial College (1983).

[Ham2] Hammond, P. and Sergot, M. "A Prolog shell for logic based expert systems",Proc. 3rd BCS Expert Systems Conference (1983).

[Hay] Hayes-Roth, F., Watermann, P.A. and Lenat, D.B.,"Building expert systems",Addison Wesley, Reading MA, 1983.

[Kow1] Kowalski, R.,"Predicate logic as a programming language",Proc. IFIP, North Holland, 1974.

[Kow2] Kowalski, R. "Algorithm = Logic+Control",*Comm. ACM*, 22, 7 (July 1979), pp. 424-436.

[Kow3] Kowalski, R.,"Logic for Problem Solving", North Holland, Amsterdam, 1979.

[Pan] Panaroni, P.,"Reti di vincoli: proprietà ed applicazioni", Tesi di Laurea, Pisa 1978.

[Per] Pereira, L.M.,"Logic control with logic",*Proc. First International Logic Programming Conference* (1982).

[Sterl] Sterling, L.S.,"Expert system = knowledge + meta-interpreter", Technical Report (1984).

IL PROGETTO ALPES
APPROCCIO E OBIETTIVI DELL'ENIDATA

S. Ghelfo, E.G. Omodeo, F. Russo, A. Torchi

ENIDATA - Divisione Prodotti TEMA - Bologna

ABSTRACT

Vengono descritte le attivita' che Enidata sta svolgendo nel progetto Esprit denominato ALPES: varie realizzazioni nel campo della deduzione automatica, rassegna e confronto dei Prolog esistenti, strumentazione del Prolog.

Introduzione

Si e' da poco concluso il progetto Esprit n.363, della durata di un anno, che e' servito semplicemente a formulare un progetto di piu' ampio respiro (Esprit n.973), che andasse a colmare una necessita' molto sentita dai cultori della Programmazione Logica, e cioe' che venga messo a punto un ambiente integrato che sia qualcosa di piu' che una semplice raccolta di strumenti di ausilio al programmatore. Sia il progetto pilota che la sua continuazione sono stati denominati "Advanced Logical Programming EnvironmentS" (in breve ALPES) (*).

L'obiettivo principale e' di giungere alla messa a punto di prototipi, implementati per lo piu' in Prolog e finalizzati a potenziare Prolog stesso. Lo stesso interprete che servira' alla prototipazione, sara' arricchito in varie direzioni (ad es. con l'aggiunta di primitive di sincronizzazione). Le tematiche piu' avanzate che si intendono approfondire nel progetto sono quelle del meta-ragionamento, e della Programmazione Automatica sotto i diversi aspetti di sintesi, validazione logica rispetto a specifiche date, trasformazioni che preservano la correttezza. Nel progetto ALPES vi e' dunque ampio spazio, oltre che per realizzazioni direttamente legate al Prolog, anche per ricerche riguardanti la Deduzione Automatica. L'Enidata si sta muovendo in entrambe le direzioni.

(*) Questo progetto vede coinvolti, assieme all'Enidata, l'Universita' di Roma (DIS), il CNR (IASI - Roma), l'Universita' di Bologna (DEIS), due laboratori del CNRS (LRI - Parigi Orsay, LSI - Toulouse), le societa' francesi Bull e Cril, due universita' tedesche (TUM - Monaco, Bundeswehr), e le Universita' di Orleans e di Lisbona.

Realizzazioni legate alla Deduzione Automatica

Come attivita' di ricerca e sperimentazione legate alla Deduzione Automatica stiamo realizzando un sistema, chiamato PRED, che raccoglie una serie di strumenti legati in parte al calcolo predicativo inferiore, in parte alla teoria degli insiemi.

PRED e' implementato in SETL, un linguaggio procedurale che grazie alle sue primitive di livello molto elevato e' particolarmente adatto per la rapida messa a punto di prototipi di sistemi software di ampie dimensioni ed eterogenei. Tuttavia le procedure costituenti PRED possono essere considerate come falsariga accurata per implementazioni da realizzarsi in Prolog.

Dato che PRED contiene, fra le altre cose, un interprete per il Prolog (puro), prepara il terreno per l'integrazione del meccanismo inferenziale del Prolog con altri metodi deduttivi; in particolare un'integrazione con derivati del procedimento di Knuth-Bendix appare molto promettente per potenziare l'applicabilita' di Prolog al campo della manipolazione algebrica simbolica.

Attualmente, PRED non e' concepito per essere un sistema dimostratore di teoremi o un verificatore della correttezza di dimostrazioni; e' semplicemente una libreria di algoritmi legati al campo della Deduzione Automatica, rivestiti quel tanto che serve per poterli collaudare, misurarne e confrontarne l'efficienza, ottenere tracciati delle esecuzioni (molto utili, questi ultimi, per rivelare il funzionamento degli algoritmi a persone da addestrare). Queste finalita' giustificano certe limitazioni del sistema nella sua forma corrente, quali l'assenza di procedure di Skolemizzazione ed una grammatica piuttosto rudimentale che accetta solo forme clausali con gli atomi scritti in notazione funzionale.

Un vantaggio che si ottiene mettendo assieme vari metodi di deduzione che in apparenza hanno solo tenui connessioni uno con l'altro, e' che molte parti del software (ad es. gli algoritmi di unificazione, o il calcolo proposizionale) sono condivise da vari sottosistemi. Di conseguenza, la manutenzione risulta piu' facile; inoltre nel collaudare un sottosistema si possono scoprire degli errori che pregiudicherebbero anche il funzionamento di altri sottosistemi. Per di piu', lo sforzo investito per ottenere un ambiente unificato per i vari sottosistemi comporta l'individuazione delle primitive piu' appropriate per implementare una ricca gamma di metodi dimostrativi.

Attualmente, PRED ha sei sottosistemi fondamentali:

- metodi per decidere la soddisfacibilita' proposizionale di forme congiuntive normali (dualmente, la validita' di forme disgiuntive);
- un interprete per il Prolog puro, che dovra' servire come base per estensioni che consentano di sperimentare vari tipi di nuove primitive;

- un dimostratore di teoremi per il calcolo predicativo, basato su di un metodo chiamato Linked Conjunct;
- un decisore di soddisfacibilita'/validita' per uguaglianze e disuguaglianze prive di variabili, basato sul metodo di Oppen;
- il procedimento di completamento/canonizzazione di Knuth-Bendix, per teorie equazionali non commutative;
- un decisore, chiamato SYLL, per formule insiemistiche non quantificate.

Il primo di questi sottosistemi, cioe' il decisore proposizionale, sviluppa (oltre al metodo delle tavole di verita') tre approcci:

. il procedimento di Davis-Putnam;
. il metodo di connessione di Bibel;
. il forward chaining, generalizzato a clausole non di Horn mediante la suddivisione in casi.

Contiene inoltre, in forma larvale, altri metodi come la risoluzione, la "locked resolution", i tableaux di Beth. Estensioni al sottosistema proposizionale saranno fatte nelle direzioni del backward chaining (combinato con la suddivisione in casi), e del trattamento diretto di forme negative normali per mezzo dei tableaux di Beth, del metodo di connessione, e di generalizzazioni dell'algoritmo di Davis-Putnam.

Il procedimento di Knuth-Bendix e' basato sulla descrizione di questo metodo contenuta nell'articolo originario. Il criterio seguito, ad ogni passo, per la scelta della prossima uguaglianza da orientare, e' che il suo peso sia minimo. Questo sottosistema di PRED e' stato implementato dapprima in C-Prolog. I motivi per tradurre in SETL la versione Prolog, sono stati:

. da un lato, come abbiamo spiegato sopra, l'esigenza di uniformita';
. d'altro lato, la prospettiva di introdurre con maggior facilita' delle ottimizzazioni nella versione SETL piuttosto che in quella Prolog.

L'implementazione Prolog ha subito varie estensioni, che mirano ad inglobare i piu' recenti e potenti metodi della teoria della riduzione.

Il sottosistema Linked Conjunct e' basato sulla documentazione relativa ad un'implementazione fatta parecchi anni fa ai Bell Laboratories. Come prototipo per dimostratori di teoremi piu' sofisticati, il metodo Linked Conjunct sembra preferibile ad un approccio basato sulla Risoluzione, dato che mantiene disaccoppiati i due processi di base:

ricerca di uno schema di unificazioni,
ricerca di una contraddizione proposizionale.

Di conseguenza, ogni nuovo progresso fatto in relazione al problema della decisione per la soddisfacibilita' proposizionale si traduce piu' immediatamente in un miglioramento del Linked Conjunct che non della Risoluzione. Si puo' inoltre sostenere che la correttezza di molti recenti raffinamenti del metodo di Risoluzione si dimostra piu' facilmente se questi sono visti come raffinamenti

del metodo del Linked Conjunct.
Per limitare lo spazio di ricerca di una refutazione, la nostra implementazione del metodo Linked Conjunct richiede che sia indicata per ciascuna clausola una molteplicita' intera, che stabilisce quante volte la clausola puo' essere adoperata. Un'altra maniera di contenere lo spazio di ricerca, puo' essere quella di fissare un limite per la complessita' degli unificatori: se l'unificazione di due termini richiede un unificatore troppo complesso, i due termini vengono trattati come se non fossero unificabili.

SYLL e' un decisore della soddisfacibilita' di formule insiemistiche non quantificate nelle quali compaiano, oltre ai connettivi proposizionali, solo i costrutti binari di appartenenza, identita', inclusione, intersezione, differenza e unione. Numerosi altri costrutti, la cui decidibilita' e' nota o congetturata verranno via via aggiunti. Fra questi, i costrutti unari che denotano le operazioni di singoletto, unione generale, e insieme delle parti. Il nostro metodo di decisione non si limita a stabilire se le formule date siano soddisfacibili o meno, ma e' anche in grado di produrre un modello insiemistico per ogni formula soddisfacibile. Inoltre SYLL potrebbe essere potenziato in modo da fargli produrre, per ogni formula, una collezione esaustiva di schemi di modello mutuamente esclusivi. Attualmente SYLL non e' stato ancora interamente amalgamato con il resto di PRED.

Rassegna dei Prolog esistenti

In Alpes e' stata fatta una rassegna dei sistemi Prolog attualmente esistenti, ponendo particolare attenzione alle primitive di controllo e alle caratteristiche impure presentate da ciascuno di essi. Una delle motivazioni di questo studio e' di poter contribuire alla standardizzazione degli aspetti extralogici del linguaggio.

Lo studio delle primitive di controllo attualmente disponibili e' stato centrato sui seguenti punti:

1. Cut (ricerca di una standardizzazione) e sue varianti.
   La definizione originaria del Prolog forniva solamente la nota primitiva di "cut"; si e' a lungo discusso su come trovare un modo migliore di controllare la strategia di valutazione standard del Prolog. L'approccio piu' positivo e' stato quello di nascondere l'uso del "cut" in primitive di controllo dal punto di vista logico piu' significative (not, once, forall, if-then-else), che sono ormai fornite dalla maggior parte dei sistemi Prolog.
2. Negazione.
   La ricerca di metodi generali ed efficienti per esprimere fatti negativi nel Prolog e' ancora in corso. Tutti i sistemi Prolog forniscono la negazione intesa come "negation as failure", con l'implicita assunzione dell'ipotesi di "mondo chiuso". Solo alcuni sistemi (IC-Prolog, MU-Prolog, Prolog II, etc.) ne danno un'implementazione corretta, controllando l'istanziazione delle variabili nel goal negato.

3. Gestione degli errori a tempo di esecuzione
   I sistemi Prolog piu' datati rispondono agli errori a tempo di esecuzione semplicemente facendo fallire il predicato in errore o facendo abortire l'esecuzione.
   Questo approccio non e' accettabile in parecchi casi: i sistemi piu' recenti danno all'utente la possibilita' di definire un gestore degli errori a cui viene ceduto il controllo trasmettendogli come parametri l'indicazione del tipo di errore e una rappresentazione del goal tentato.
4. Controllo sulla regola di calcolo
   Con la regola di calcolo standard del Prolog (valutazione da sinistra a destra e in profondita'), l'unica possibilita' di esercitare un controllo sulla valutazione si ha mediante l'ordinamento dei goals all'interno delle clausole. Questa rigidita' pone delle limitazioni: da una parte un determinato ordinamento risulta efficace solo per un particolare uso della clausola; dall'altra si e' costretti a mescolare aspetti di controllo con aspetti piu' propriamente dichiarativi. Per dare maggiore flessibilita' e consentire piu' chiarezza ai programmi senza limitarne l'efficenza ci si muove in due direzioni:
   strategie di coroutinaggio,
   pieno sfruttamento del parallelismo.
   Nel primo approccio il controllo viene espresso attraverso annotazioni sulle variabili dei goals, che rendono questi o produttori "pigri" o consumatori "avidi" dei valori istanzianti le variabili. Un goal che sia consumatore rispetto ad una variabile non e' abilitato ad istanziarla, e viene sospeso nel momento in cui tenta di farlo perche' ha bisogno di un'ulteriore parziale istanziazione della stessa. Il controllo ritorna ad un goal consumatore nel momento in cui qualche produttore rende disponibile una nuova istanziazione. Un meccanismo analogo si ha per i goals che sono produttori. Questa soluzione e' adottata in sistemi ancora fondamentalmente sequenziali (IC-Prolog, MU-Prolog, Prolog II, etc.).
   Nel secondo approccio si cerca di sfruttare appieno sia il parallelismo or che il parallelismo and, eventualmente privilegiando l'uno o l'altro a seconda del tipo di sistema. Anche in questo caso il controllo si esercita in genere utilizzando annotazioni che rendono le variabili canali mono-direzionali di dati. Sistemi di questo tipo sono il Concurrent Prolog e il Parlog.
5. Primitive di meta-livello
   Tutti i sistemi Prolog rivolti ad applicazioni offrono primitive che permettono all'utente di trattare formule e termini, in particolare i passi del processo di deduzione, come oggetti del discorso.
   Queste primitive rendono possibile le realizzazione di meta-interpreti attraverso i quali un sistema Prolog puo' essere esteso in molte direzioni. Fra le altre:
   - realizzazioni di sintassi piu' orientate all'utente;
   - arricchimento del potere espressivo;

- realizzazioni di diversi meccanismi di inferenza (forward chaining, strategie di ricerca, etc.);
- strumenti di spiegazione e monitoraggio;
- programmazione top-down, type checking e modularizzazione della base delle asserzioni;
- valutazione parziale;
- algoritmi di tracing e di debugging intelligente.

Alcuni recenti sistemi di programmazione logica (Metalog, Amalgama, etc.) dispongono di un meta-interprete che permette all'utente di intervenire in ogni passo del processo di deduzione tramite informazioni espresse in meta-clausole. Questa informazione di controllo puo' essere vista come conoscenza di meta-livello cioe' su come usare in maniera efficiente la conoscenza disponibile a livello base realizzando delle strategie euristiche per la soluzione di una data classe di problemi.

Strumentazione del Prolog

Una delle linee del progetto ALPES e' orientata allo sviluppo di strumenti di ausilio alla programmazione e di trasformazione dei programmi, come primi passi verso la definizione di un ambiente integrato. In quest'ambito l'Enidata sta lavorando alla realizzazione di un valutatore parziale, cioe' di un algoritmo di trasformazione di programmi che puo' essere usato in alternativa (o anche congiuntamente) ad un compilatore per aumentare l'efficienza dei programmi. Uno strumento di questo tipo consente al programmatore di curare soprattutto la chiarezza e la correttezza di programmi, magari utilizzando tecniche di programmazione strutturata come i tipi di dato astratti, senza preoccuparsi eccessivamenti dei problemi di efficenza.

Il nostro valutatore parziale e' stato sviluppato partendo da un meta-interprete e utilizzando principalmente tre tecniche di ottimizzazione statica: istanziare i parametri di un programma propagando i valori degli argomenti attraverso il programma, ridurre il numero di inferenze logiche espandendo l'invocazione di un predicato con le clausole rilevanti della sua definizione, e valutare i predicati builtin ogniqualvolta e' possibile.

Sono stati inoltre sviluppati in Enidata (parzialmente anche nell'ambito del progetto Esprit No.530), dei prototipi di algoritmi per il tracing e il debugging.

Il nostro procedimento di tracing offre diversi vantaggi rispetto ai tracers comunemente messi a disposizione dai vari sistemi Prolog:
- vengono mostrati separatamente la chiamata di un goal ed il matching con una clausola, che viene identificata tramite il suo numero d'ordine tra quelle definite per quel dato predicato;

- vengono distinti chiaramente i predicati di sistema da quelli definiti dall'utente;
- vengono date maggiori informazioni sulle modalita' di successo o di fallimento di un goal (ad es. se un goal fallisce perche' il relativo predicato non e' definito, se fallisce per via di un cut, o se un goal e' stato soddisfatto usando una clausola o un fatto);
- la valutazione viene seguita anche all'interno di primitive come il not o l'if-then-else;
- sono disponibili comandi che consentono, oltre che di tracciare solo l'esecuzione di predicati su cui sono stati posti degli spy-points, anche di sorvolare sulla valutazione del corpo di una clausola (mostrando solo il risultato finale) ed inoltre di chiedere la ripetizione della valutazione di un goal.

Non e' ancora stata introdotta la possibilita' di vedere le variabili con i loro nomi originari, ne' vi sono primitive per la gestione del video.

Nel campo degli strumenti per il debugging l'Enidata ha sviluppato un primo prototipo, che implementa gli algoritmi descritti da E. Y. Shapiro (in "Algorithmic Program Debugging"), introducendo anche alcune estensioni che permettono di trattare programmi Prolog comprendenti primitive come cut, not, if-then-else, or.

La strategia di debugging adottata da questo prototipo consiste nell'eseguire, tramite un meta-interprete, il programma che si comporta in maniera errata, confrontando in maniera sistematica il suo comportamento effettivo con quello corretto - nel senso definito dall'utente - nei punti rilevanti dell'esecuzione.

Ragionando sulle differenze rilevate tra il comportamento effettivo del programma e quello atteso, gli algoritmi di diagnosi sono in grado di localizzare delle clausole errate, spesso dando anche delle indicazioni per focalizzare il tipo di errore nella clausola.

Questi algoritmi sono in grado di diagnosticare errori in corrispondenza a tre tipi di comportamento errato di un programma:
. il programma termina dando un risultato errato,
. il programma non da' un risultato richiesto,
. il programma va in ciclo.

All'utente vengono rivolte delle domande necessarie per stabilire quale dovrebbe essere il comportamento corretto del programma, e che ricadono in tre categorie, a seconda del tipo di errore diagnosticato:
. se un goal e' stato risolto correttamente o meno,

. se un goal e' risolubile e, in tal caso, qual'e' l'insieme delle sue soluzioni,
. se e' corretto che, per la risoluzione di un dato goal, venga chiamato un certo altro goal.

Partendo da questo primo prototipo si sta ora lavorando ad un suo miglioramento, seguendo anche l'approccio di L. M. Pereira al "Rational Debugging", nelle seguenti direzioni:

- minimizzazione del numero di domande che vengono rivolte all'utente: dovrebbero idealmente essere poste solo le domande strettamente necessarie all'identificazione dell'errore. Qusto si puo' ottenere, ad esempio, seguendo all'indietro i passi che hanno portato alla costruzione di termini errati nel risultato di una valutazione.
- Eliminazione delle domande che richiedono di fornire l'insieme delle soluzioni di un goal; l'utente dovrebbe rispondere solo a domande piu' semplici circa la correttezza di una soluzione, la risolvibilita' e ammissibilita' di un goal o la completezza di un insieme di soluzioni.
- Miglior trattamento dei programmi che contengono primitive 'cut'.
- Possibilita' di dare informazioni piu' precise sul tipo di errore rilevato, non solo l'identificazione di una clausola errata o mancante.

Come Implementare Un Prolog Veloce Su Microcalcolatori A Basso Costo

Laurence Archer
SPL Italia

In questa presentazione analizzeremo cio' che occorre per implementare un Prolog veloce ed efficiente avente lo scopo di ottenere alte prestazioni su hardware a basso costo. Illustreremo varie alternative che permettono di soddisfare requisiti come risoluzione ad alta velocita' e basso turnover di memoria, esigenze fondamentali sia per l'utente industriale che accademico. In questa direzione si potrebbero avere prestazioni eccellenti dalle architetture parallele e dalle macchine basate su microcodice pero' cio' significherebbe ricorrere all'impiego di macchine costose ed altamente specializzate. Oggi, invece, l'utenza industriale necessita prima di tutto di un sistema Prolog in grado di effettuare alcune migliaia di inferenze logiche al secondo (LIPS) su macchine di basso costo e facilmente reperibili come il PC IBM.

Discuteremo inoltre le carenze di molti degli interpreti Prolog a basso costo disponibili sui microcalcolatori alla luce dell'origine degli interpreti stessi e della tecnologia con cui sono implementati. I tools di questo tipo sono in linea di massima poco proponibili come strumenti di produzione di livello industriale a causa delle loro basse prestazioni, del loro alto turnover di memoria, della mancanza di data-types industriali (ad esempio stringhe di caratteri e numeri a virgola mobile) e della rozza interfaccia uomo macchina di cui sono dotati. Se si vuole che Prolog diventi il linguaggio standard per le applicazioni commerciali basate sulle tecniche di intelligenza artificiale questi problemi dovrannno essere superati dalla prossima generazione di sistemi Prolog compilati.

Illustreremo quindi i requisiti base di un compilatore Prolog nei termini delle operazioni primitive coinvolte nell'esecuzione di un programma Prolog: unificazione, definizione del goal (chiamata ad una procedura), backtracking e "cut". Viene esplorato il nucleo del progetto di una macchina virtuale che implementa un insieme astratto di istruzioni per Prolog. La struttura di massima della macchina Prolog astratta viene discussa in termini di registri e di stacks, insieme con i tipi di istruzione che vengono eseguiti.

Verranno "compilati a mano" per la macchina astratta alcuni semplici programmi Prolog, per dare un'idea del codice generato e per mostrare alcune delle ottimizzazioni possibili. Discuteremo poi come implementare un compilatore ottimizzante per la macchina virtuale Prolog.

Infine verra' messa in relazione la macchina virtuale presentata con i vincoli imposti dai computer per l'ufficio disponibili commercialmente, in particolare con quelli imposti dall'architettura del PC IBM. Daremo inoltre un'idea delle prestazioni e dei fabbisogni di memoria relativi ad una implementazione commerciale avanzata di Prolog basata su questa tecnologia.

IL PROLOG E LA PROGETTAZIONE DI BASI DI DATI

Georg Gottlob (')
Letizia Tanca ('')

(') Istituto per la Matematica Applicata - CNR Genova
('') Politecnico di Milano e Universita' di Napoli

ABSTRACT

Un programma in Prolog per la normalizzazione delle relazioni di una base di dati presenta una serie di vantaggi rispetto a programmi dello stesso tipo scritti in linguaggi tradizionali: il Prolog e', per la sua struttura, molto adatto a rappresentare sia la conoscenza statica della base di dati che l'informazione semantica; inoltre, sue caratteristiche sono anche l'estrema concisione e la leggibilita'.
In questo lavoro viene esaminato il problema della rappresentazione in Prolog della conoscenza semantica relativa a uno schema di base di dati. Viene inoltre presentato un programma per la riduzione di relazioni in quarta forma normale, per l'eliminazione cioe' di dipendenze multivalore indesiderate da uno schema relazionale originariamente asserito.

INTRODUZIONE

Tra i diversi modelli di rappresentazione di una base di dati [1], il modello relazionale [3,4] si e' rivelato il piu' promettente. La maggior parte dei database systems sviluppati di recente,come ad esempio DB2 (IBM) [2], Ingres [5], e quasi tutti quelli concepiti per micro e personal computers si basano sul modello relazionale. Le ragioni piu' importanti del successo di tale modello sono la sua semplicita' concettuale e il fatto che sia molto ben formalizzato in termini matematici.
Nel progetto di una base di dati relazionale e' possibile operare scelte molto differenti rispetto al modo di raggruppare i dati a formare i diversi schemi di relazioni, e, per varie ragioni, alcune scelte risultano piu' convenienti di altre.
Un concetto fondamentale nella progettazione di schemi di basi di dati e' quello di **dipendenza**, che e' un vincolo semantico su quali relazioni possono essere assunte come valori di un certo schema relazionale. Le dipendenze possono causare dei problemi nella rappresentazione della base di dati e nella sua gestione, ad esempio ridondanze, oppure "anomalie",come inconsistenze etc.
Le forme normali sono state introdotte nella progettazione di basi di dati proprio per evitare questo tipo di problemi. La teoria e' abbastanza ben sviluppata [6,7], ed e' stata anche sperimentata nell'effettivo progetto di basi di dati [8].Gli algoritmi e le tecniche per la riduzione in forma normale prodotti negli ambienti di ricerca , comunque, non sono molto diffusi all'esterno, ed e' percio' auspicabile l' implementazione di strumenti software per il supporto della progettazione di basi di dati, cosi' da sperimentare l' efficacia della teoria anche fuori dalla comunita' degli informatici.

Lo scopo di questo lavoro e' mostrare come il linguaggio di programmazione Prolog sia adatto per l'implementazione di tali strumenti.
Faremo riferimento a un lavoro precedente [11] dove il problema della normalizzazione e' stato affrontato con particolare attenzione a un certo tipo di dipendenze -le **dipendenze funzionali** - e dove sono stati presentati anche alcuni altri algoritmi per determinare elementi caratteristici della base di dati, come le **chiavi** di una relazione o la **chiusura** di un certo insieme di

dipendenze funzionali. Questi algoritmi sono stati integrati per formare il nucleo di un sistema atto alla progettazione di piccole basi di dati. Il sistema, sviluppato in Prolog, fornisce algoritmi per la riduzione in terza forma normale (3NF) e in forma normale di Boyce-Codd (BCNF), per la produzione cioe' di relazioni libere da dipendenze funzionali indesiderate.
In questo lavoro ci occupiamo del problema delle **dipendenze multivalore**, e presentiamo un programma in Prolog che effettua la riduzione di relazioni in quarta forma normale, cioe' che decompone gerarchicamente schemi di relazioni fino a ottenere nuovi schemi liberi da dipendenze multivalore indesiderate.

In un sistema software per la progettazione automatica di basi di dati devono essere incorporati la struttura statica della base di dati (schema di ogni relazione), la conoscenza semantica e i vincoli (come le dipendenze) e un certo numero di algoritmi per modificare la struttura della base di dati (ad esempio gli algoritmi di normalizzazione).
La ragione principale per cui noi sosteniamo la scelta del Prolog come linguaggio adatto a questo scopo e' che esso permette in modo naturale una descrizione integrata di questo tipo: le strutture dei dati possono essere rappresentate da fatti in Prolog, la conoscenza semantica da fatti oppure da regole e gli algoritmi da regole.
Inoltre, molti problemi tipici della progettazione di basi di dati possono essere espressi come formule logiche precedute da quantificatori esistenziali e universali, che sono facilmente traducibili in clausole di Horn.

Scelte fondamentalmente differenti possono comunque essere effettuate relativamente alla descrizione della base di dati e del problema. Ad esempio, le dipendenze funzionali (fd) si lasciano rappresentare in due modi ortogonali, che danno origine ad approcci totalmente diversi alla risoluzione del problema della normalizzazione.
In [11], si e' scelto di rappresentare le dipendenze funzionali come fatti. In questo lavoro viene mostrato,invece, che rappresentarle come regole permette di sfruttare in pieno il potere inferenziale del Prolog. Si nota anche, pero', che la stessa descrizione non e' altrettanto fattibile nel caso delle dipendenze multivalore (mvd), e percio', per ottenere uniformita' di trattazione, anche qui ci si atterra' alla prima scelta.

1.RAPPRESENTAZIONE IN PROLOG DELLA STRUTTURA DELLA BASE DI DATI E DELLA CONOSCENZA SEMANTICA

In una base di dati relazionale, la "conoscenza statica" consiste nell'insieme delle **relazioni** (1) .
Siccome, durante la fase di progettazione, non siamo tanto interessati a singoli attributi, quanto ad insiemi di attributi, cioe' sottoinsiemi dello schema, rappresentiamo uno schema relazionale usando la struttura a lista del Prolog.
Cosi', per esempio, lo schema di una relazione con attributi A,B,C,D puo' essere asserito come segue:

```
schema ([a,b,c,d]).
```

Si noti comunque che i nomi degli attributi formano un insieme ordinato, piu' che una vera lista, nel senso che un attributo non e' mai presente piu' di una volta nello stesso schema.

La riduzione in forma normale consiste nella decomposizione iterativa di uno schema relazionale in sottoschemi.
Nel nostro caso, quello della riduzione in quarta forma normale (4NF), l'algoritmo prende in ingresso lo schema originale della relazione, e lo sostituisce con due sottoschemi contententi un numero minore di attributi,

(1) Per la definizione di relazione e di schema relazionale, vedi Ullman [6].

continuando ricorsivamente con questi due fino a quando tutti gli schemi sono in 4NF.

L' altro problema di rappresentazione al quale siamo interessati e' quello della "conoscenza semantica", cioe' delle dipendenze che sono nella base di dati.Ci occuperemo soltanto di dipendenze funzionali e multivalore (2) .
E' noto [11] che ogni dipendenza funzionale puo' essere sostituita equivalentemente da un insieme di dipendenze funzionali che hanno un solo attributo nella parte destra. Cio' rende possibile riscrivere qualsiasi dipendenza funzionale come un insieme di clausole di Horn, e quindi di regole Prolog, dove il lato destro della dipendenza diventa l'antecedente della regola. Cosi', una dipendenza A->B,C diventa la coppia di regole:

```
b:-a.   ,   c:-a.
```

il che permette di sfruttare la forte analogia tra l'implicazione della logica proposizionale e dipendenze funzionali (3) .

Questa forma permette di usare il potere inferenziale del Prolog per dedurre che una data dipendenza segue logicamente dall'insieme di quelle asserite (4) . Cosi', dato un insieme di fd scritte come regole Prolog, sara' facile dedurre se una certa altra regola (fd) ne segue mediante il seguente programma:

```
determines (X,Y) :- asserta(X), Y, retract(X), !.
determines (X,Y) :- !, retract(X), fail.
```

L'altra possibilita' e' di rappresentare le dipendenze funzionali come fatti Prolog. Ad esempio, una dipendenza A->B,C sarebbe asserita come:

```
fd([a],[b,c]).
```

Quest'ultima rappresentazione e' quella scelta in [11].
Sebbene la prima forma sia piu' elegante dal punto di vista di un logico, la seconda e' piu' appropriata. Infatti, le dipendenze multivalore (mvd) non si possono rappresentare come clausole di Horn , poiche' il contenuto semantico di una mvd con piu' di un attributo nella parte destra viene radicalmente modificato dalla sua sostituzione con un insieme di dipendenze che hanno al lato destro un solo attributo; percio', per ottenere un trattamento uniforme di entrambi i tipi di dipendenza si preferisce la seconda possibilita' di rappresentazione.
La rappresentazione in Prolog di una dipendenza multivalore X->->Y,Z,W sara' percio' del tipo:

```
mvd([x],[y,z,w]).
```

Per capire le implicazioni logiche tra dipendenze, e' utile avere delle regole che permettono di sapere come una o piu' dipendenze implicano altre dipendenze, cioe' di computare la chiusura di un insieme dato di dipendenze. Oltre ai cosiddetti "Assiomi di Armstrong" , che formano un insieme valido e

(2) Per le relative definizioni rimandiamo ancora a Ullman [6].

(3) L'equivalenza tra le fd e la logica proposizionale e' stata dimostrata in [9].

(4) L'insieme di tutte le dipendenze che seguono logicamente da un dato insieme di dipendenze D viene detto **chiusura** di D e indicato con D^+. E' noto che, se D e' un insieme di fd, e D^* e' l'insieme corrispondente di implicazioni logiche,
A -> B e' in D^+ sse D^* |- (A -> B)*.

completo di regole di inferenza per le dipendenze funzionali, sono state trovate da Beeri, Fagin and Howard [10] altre regole di inferenza per un insieme di dipendenze funzionali e multivalore su un certo insieme di attributi, che formano anch'esse un insieme valido e completo. Di nuovo, siccome questi assiomi sono essenzialmente clausole di Horn, possono facilmente essere tradotti in Prolog.

2. DECOMPOSIZIONE IN QUARTA FORMA NORMALE

La quarta forma normale si applica a relazioni con dipendenze sia multivalore che funzionali. E' una generalizzazione della forma normale di Boyce Codd (BCNF), nel senso che, se una relazione S contiene solo dipendenze funzionali, essere in 4NF significa per S essere in BCNF.
Sia S uno schema di relazione e D un insieme di dipendenze che valgono in S. Si dice che S e' in **quarta forma normale** rispetto a D se, dati due insiemi di attributi di S, L ed R, non appena c'e' una mvd L ->-> R,con R non vuoto ne' incluso in L, e LR non contiene tutti gli attributi di S, allora L contiene una chiave per S.

La decomposizione di relazioni in quarta forma normale e' un modo molto potente per eliminare pressoche' tutta la ridondanza presente in una base di dati.
L'algoritmo piu' ovvio [6] per ridurre una relazione in 4NF rispetto a un insieme di dipendenze D usa la chiusura D^+ dell'insieme D, e percio' risulta molto lento. Infatti, computare la chiusura di un insieme di dipendenze e' di per se' altamente esponenziale nella cardinalita' di D. Percio', sebbene noi abbiamo implementato anche questo algoritmo, lo troviamo poco soddisfacente per il nostro scopo.

Esistono altri algoritmi che sono solo polinomiali [12], ma sono altrettanto insoddisfacenti, visto che non verificano che una relazione sia gia' in 4NF, percio' ci si puo' ritrovare a decomporre relazioni che sono gia' nella forma desiderata ; di fatto, il solo problema di tale verifica risulta NP-completo [13]!
Il nostro algoritmo usa invece il concetto di **base di dipendenza** di un insieme di attributi.

Dato un insieme di attributi X di uno schema relazionale S, e l'insieme delle sue dipendenze funzionali e multivalore D, e' possibile ottenere una partizione di S-X in un numero finito di insiemi di attributi Z_i tali che, se Y e' contenuto in S-X, allora X->->Y e' in D se e solo se Y e' unione di alcuni degli Z_i. Chiamiamo l'insieme degli Z_i la **base di dipendenza** di X rispetto a D.

Nel nostro algoritmo, piuttosto che costruire la chiusura di D, preferiamo verificare se una data dipendenza X->->Y vale,determinando semplicemente la base di dipendenza di X e vedendo se Y-X e' unione di alcuni dei suoi insiemi (5) .
Daremo ora una breve descrizione dell'algoritmo.

Sia U lo schema relazionale che si vuole originariamente decomporre. Sia D l'insieme delle dipendenze,contenente sia fd che mvd.

a. Se in D esistono delle fd, vengono riscritte come mvd (ad esempio,se c'e' A->B,C devono essere asserite le due mvd A->->B e A->->C) (6) .

b. Sia S lo schema da decomporre in questo passo. Si verifica se S e' gia' in 4NF.

(5) Questa verifica impiega solo un tempo polinomiale.

(6) Una delle regole di inferenza di Beeri, Fagin e Howard assicura che le fd sono casi speciali di mvd [6].

c. Se S non e' in 4NF, ed esiste una mvd L->->R in D tale che L ed R sono contenuti in S, R e' non vuoto e non e' un sottoinsieme di L, LR non ricopre tutto S, ed L non e' una chiave per S, allora si decompone S in S_1=S-R e S_2=LR usando questa dipendenza (vedere Ullman [6] per dettagli su questa decomposizione), e poi si chiama ricorsivamente la decomposizione di S_1 e S_2 tornando al passo 2.

d. Se tale dipendenza non esiste in D, si verifica se puo' essere generata a partire da D:

   d.1. Si prende un sottoinsieme X di S, e un sottoinsieme Y di S-X, tali che X non sia una chiave per S e XY non sia tutto S.

   d.2. Per verificare se X->->Y, si costruisce la base di dipendenza di X e si vede se Y e' unione di alcuni dei suoi insiemi.

   Questo deve essere fatto per tutti i possibili sottoinsiemi X e Y, fino a quando si trove una tale dipendenza ,altrimenti vuol dire che S e' gia' un 4NF.

e. Se la dipendenza che viola la 4NF e' stata trovata, si decompone come al passo c. Altrimenti abbiamo raggiunto la fine del processo di decomposizione per S.

Presentiamo ora due teoremi che permettono di limitare le possibili scelte di sottoinsiemi X e Y di cui al passo d.

Faremo vedere che, all'inizio del processo di decomposizione, quando lo schema S da decomporre e' U stesso, e tutte le fd sono gia' state asserite come mvd, e' sufficiente decomporre soltanto rispetto all'insieme originario di dipendenze D. Cio' e' una conseguenza del

TEOREMA 1 : Sia U uno schema e D il suo insieme di dipendenze. Se nessuna di queste viola la 4NF, allora nessuna altra dipendenza che viola la 4NF puo' essere generata da D mediante gli assiomi di Beeri, Fagin e Howard .

PROVA : Infatti, se nessuna dipendenza di D viola la 4NF, vuol dire che tutte le dipendenze non banali hanno la parte sinistra che e' chiave per U. Ora, e' facile verificare dall'esame degli assiomi che tutte le dipendenze che possono essere derivate da queste hanno anch'esse la parte sinistra che e' chiave per S.-c.v.d.

Percio', se U non e' in forma normale, deve esistere almeno una dipendenza di D che la viola, e percio' si puo' decomporre rispetto a questa dipendenza. Naturalmente, questo teorema puo' essere applicato soltanto per decomporre lo schema originale U: invero, quando lo schema S da decomporre non e' U, mvd che sono banali per U potrebbero essere non banali per S, e insiemi di attributi che sono chiavi per S potrebbero non essere chiavi per U.

Il secondo teorema e' relativo al caso generale, cioe' a quando lo schema da decomporre S non e' quello originario. In tal caso, per ogni sottoinsieme X di S, bisogna esaminare un certo numero di sottoinsiemi Y di S-X per vedere se possono essere costruiti a partire dalla base di dipendenza di X. Il teorema limita il numero di possibili "Y" che il programma deve esaminare.

TEOREMA 2 : Se X->->Y (Y non vuoto) e' una mvd che viola la 4NF (cioe' XY non e' tutto S, Y non e' contenuto in X e X non e' una chiave per S), se a e' un elemento di Y, esiste una mvd X->->Y' che viola la 4NF e tale che a non appartiene a Y'.

PROVA : Sia Y'=S-X-Y. Da X->->Y, per l'assioma "di complementazione" per le mvd, segue X->->Y'.
Occorre mostrare che anche la dipendenza X->->Y' viola la 4NF. Invero, Y' e' non

vuoto, altrimenti XY sarebbe l'intero S, Inoltre, Y' non e' contenuto in X per definizione, e XY' non e' tutto S, perche' X non contiene Y'.

COROLLARIO : Nell'algoritmo per trovare mvd che violano la 4NF, per ogni data parte sinistra L, e' sufficiente verificare le mvd L->->R tali che R e'contenuto in S-L-{a}.

4.DESCRIZIONE DEL PROGRAMMA

Presentiamo ora il programma con una breve descrizione. Lo schema originario U deve essere asserito come

origschema([a,b,.....]).

Le fd e le mvd vengono asserite nel modo indicato in precedenza.

Il "goal" principale del programma e':

```
start:- makemvds,origschema(U),decompose(U).
```

Il goal **makemvds** e' la traduzione delle fd in mvd, nel primo passo dell'algoritmo del paragrafo precedente:

```
makemvds:- fd(L,R),elem(A,R), not mvd(L,[A]), assertz(mvd(L,[A])),fail.
makemvds.
```

Questa procedura prende gli attributi della parte destra di una fd L->R uno per volta e costruisce una mvd L->->A per ogni attributo A di R.

Quindi ha luogo la decomposizione del generico schema S.

```
decompose(S):- S=[_,_],!,write(S),nl.
decompose(S):- mvd(L,R), subset(L,S), subset(R,S),not trivial(L,R,S),
               not key(L,S),!,furtherdecompose(S,L,R).
decompose(S):- origschema(U),not(S=U),subset(X,S),not(X=[]),
               not key(X,S), retractall(t(Q)),bb(X),minus([A|Tail],S,X),
               subset(Y,Tail),not(Y=[]),impliedmvd(X,Y),!,furtherdecompose(S,X,Y).
decompose(S):- write(S),nl.
furtherdecompose(S,L,R):- minus(H,S,R), union(N,L,R,S),
                          decompose(H), decompose(N).
trivial(X,Y,S):-subset(Y,X),!.
trivial(X,Y,S):-origschema(U), union(S,X,Y,U).
```

La prima clausola si riferisce al caso in cui S consta di due soli attributi: allora e' gia' in 4NF.

Nella seconda clausola, se una mvd non banale che viola la 4NF e' gia' nell'insieme di dipendenze originario D, la decomposizione ha luogo rispetto a questa mvd, e tutta l'operazione **decompose** e' ripetuta ricorsivamente sui due schemi risultanti.

La terza clausola cerca mvd che non sono in D. A causa del Teorema 1, si riferisce solo a schemi che non sono quello originario. Per prima cosa, prende un sottoinsieme non vuoto X di S, che non sia una chiave di S. Questo viene verificato con l'ausilio dei predicati:

```
key(K,S) :- closure(K,C),subset(S,C).
closure(X,RESULT) :- fd(LHS,RHS), subset(LHS,X), not subset(RHS,X),
                     origschema(U),union(W,X,RHS,U), !, closure(W,RESULT).
closure(X,X).
```

che trovano una chiave di uno schema dato S.

Il predicato **retractall** elimina qualunque cosa sia stata asserita nella ricerca della base di dipendenza in un passo di ricorrenza precedente della decomposizione. Quindi, si costruisce la base di dipendenza, basandosi

sull'algoritmo descritto in Ullman [6]. Se un insieme Z e' nella base di dipendenza viene asserito il fatto t(Z).

```
bb(X):-origschema(U),mvd(W,Y),subset(W,X),minus(Z,Y,X),assertelt(Z),
       minus(H,U,X),minus(Z1,H,Y),assertelt(Z1),fail.
bb(X):-t(Z1),t(Z2),not(Z1=Z2),inter(I,Z1,Z2),not(I=[]),minus(R1,Z1,I),
       minus(R2,Z2,I),assertelt(I),assertelt(R1),assertelt(R2),
       (( not(Z1=I),not(Z1=R1),not(Z1=R2),retract(t(Z1)));
        ( not(Z2=I),not(Z2=R1),not(Z2=R2),retract(t(Z2)))), fail.
bb(X):-mvd(V,W),t(Y),not(Y=W),
       inter(I,W,Y),not(I=[]),not(I=Y),inter(J,V,Y),J=[],
       assertelt(I),minus(H,Y,W),assertelt(H),retract(t(Y)),fail.
bb(X).
assertelt(X):-not(X=[]),not t(X),!,assertz(t(X)).
assertelt(X).
```

Quindi, basandosi sul corollario al Teorema 2, un attributo A viene escluso dall'insieme S-X: se un sottoinsieme di S-X che contiene A da luogo a una mvd che viola la 4NF, allora esiste anche una mvd "complementare" X->->(S-X-Y) che viola la 4NF, percio' si puo' decomporre rispetto a quest'ultima. Percio', solo sottoinsiemi di S-X-{A} vengono esaminati.
Il predicato **impliedmvd** verifica se il sottoinsieme Y di S-X puo' dare origine a mvd che violano la 4NF: per vedere se Y puo' essere costruito come unione di insiemi della base di dipendenza di X, e per non dover costruire tutte le unioni possibili, viene usata la seguente catena di equivalenze:
Y e' unione di insiemi Z_j della base di dipendenza di X se e solo se

$$Y^j = \bigcup_{Z_j \text{ in } Y} Z_j$$

e questo si puo' anche scrivere come

$$\forall a \in Y\ \exists Z_j \subseteq Y : a \in Z_j \iff \neg(\exists a \in Y : \neg(\exists Z_j \subseteq Y,\ a \in Z_j)).$$

Percio'

```
impliedmvd(L,R):- not existsbadx(R).
existsbadx(R) :- elem(X,R),not ( t(G),subset(G,R),elem(X,G) ).
```

Una volta trovato l'Y che si cercava, la decomposizione viene chiamata ricorsivamente dal predicato **furtherdecompose** sui due schemi risultanti. Se un tale Y non viene trovato, allora e' stata raggiunta la fine del processo di decomposizione per S, e percio' si puo' scriverne il risultato al terminale mediante l'ultima clausola **decompose.**
Qui di seguito sono elencati i programmi di utilita' che abbiamo adoperato nel nostro programma.

```
/* Predicati e operazioni su insiemi ordinati */

elem(E,L) :- L=[H|T],(E=H;elem(E T)).  /* E e' elemento di L */

subset([],X).
subset([HL|TL],[HM|TM]) :- HL=HM , subset(TL,TM).
subset([HL|TL],[HM|TM]) :- subset([HL|TL],TM).

union([],[],[] L).
union(A,A,[],L) :- not(A==[]).
union(B,[],B,L) :- not(B==[]).
union([HU|TU],[HA|TA],[HB|TB],[HL|TL]):-
     HA=HL,HB=HL,  HU=HA, union(TU,TA,TB,TL);
```



```
     HA=HL,not(HB==HL),  HU=HA, union(TU,TA,[HB|TB],TL);
     HB=HL,not(HA==HL),  HU=HB, union(TU,[HA|TA],TB,TL);
     not(HA==HL),not(HB==HL), union([HU|TU],[HA|TA],[HB|TB],TL).

minus1([],[],B):-!.
minus1(Z,[HA|TA],B):- (HA=B,TA=[],Z=[],!);
                      (HA=B, Z=TA,!);
                      (not HA=B, Z=[HZ|TZ], HZ=HA, minus1(TZ,TA,B)).

minus(R,R,[]):-!.
minus(Z,A,[HB|TB]):- minus1(R,A,HB), minus(Z,R,TB).

append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

inter([].[],X):-!.      /*intersezione*/
inter([H|T],[H|D],List):- append(L1,[H|L2],List),!, inter(T,D,L2).
inter(I,[H|D],List):- inter(I,D,List).
```

BIBLIOGRAFIA

[1] Date, C.J.,"An introduction to database systems", 3rd ed., Addison Wesley, Reading MA, 1981.

[2] Date, C.J., "A guide to DB2", Addison Wesley, Reading MA, 1984.

[3] Codd, E.F., "A relational model for large shared data banks", Communications of the ACM, 13, pp.377-387, 1970.

[4] Codd, E.F., "Further normalization of the database relational model", in R.Rustin ed., Data Base Systems Courant Computer Science Symposia, 6, Prentice-Hall, NJ Englewood Cliffs, 1972.

[5] Stonebraker, M., E.Wong, P.Kreps, and G.Held [1976]. "The design and implementation of INGRES",ACM Transactions on Database Systems.

[6] Ullman, J.D., "Principles of Database Systems", 2nd. ed., Computer Science Press, Potomac NJ, 1982.

[7] Maier, D., "Database Theory", Computer Science Press, Rockville, Md.

[8] Blanning, R.W., "Issues in the design of relational model management systems", Proc. National Computer Conference, pp.395-401, June 1983.

[12]Fagin, R., "Functional dependencies in a relational database and propositional logic", IBM Journal of Research and Development, 21. 6 (Nov 1977), pp.534-544.

[10]Beeri, C., R. Fagin and J.H. Howard, "A complete axiomatization for functional and multivalued dependencies", ACM SIGMOD International Symposium on Management of Data, pp.47-61, 1977.

[11]Ceri, S. and G. Gottlob, "Normalization of relations and Prolog", in stampa su Communications of the ACM.

[12]Tsou, D.M., and P.C.Fisher, "Decomposition of a relation scheme into Boyce Codd normal form", ACM-SIGACT, pp.23-29, 14:3, Summer 1982.

[13]Beeri, C., and P.A. Bernstein, "Computational problems related to the design of normal form relation schemes." ACM Transactions on Database Systems 4:1, pp.30-59, 1979.

CHIUSURE TRANSITIVE NELLE BASI DI DATI LOGICHE

B. Demo

Dipartimento di Informatica, Universita' di Torino
v. Valperga Caluso 37 ▫ 10125 TORINO

L'uso della programmazione logica per accedere ad una base di dati permette di dedurre nuovi fatti da quelli effettivamente presenti nella base di dati. Nel caso in cui le regole di derivazione siano ricorsive, il processo di derivazione stesso consiste nel calcolare il punto fisso della trasformazione associata alle regole. Si pone allora il problema di ottimizzare tale calcolo. In questo lavoro viene fatta una breve analisi di alcune proposte. Vengono poi espresse delle considerazioni su variazioni algebriche del processo di derivazione che rendono possibili ottimizzazioni del medesimo nel caso di clausole che definiscono una chiusura transitiva.

1. INTRODUZIONE

Le recenti ricerche sull'uso della programmazione logica per accedere ad una base di dati hanno tra le altre motivazioni quella di mettere a disposizione degli utenti un linguaggio di accesso piu' potente di quelli tradizionali che permetta di esprimere operazioni non strettamente inerenti la base di dati (quindi un linguaggio di programmazione generale) e sia capace di dedurre nuovi fatti da fatti presenti nella base di dati (in modo piu' potente di quanto possano fare sistemi convenzionali attraverso il meccanismo delle "viste di utente") [Chan78, Chan81, Chak82, Gall83].

1.1. Basi di dati logiche

In una base di dati logica o deduttiva, descritta mediante clausole di Horn, distinguiamo predicati che definiscono relazioni di base o reali, cioe' relazioni esplicitamente memorizzate nella base di dati su memoria secondaria, e relazioni derivate o virtuali, definite in termini di relazioni di base o altre relazioni virtuali.

Come esempio si consideri la relazione virtuale "antenato" definita usando la relazione reale "padre" e la stessa relazione "antenato" come segue:

```
antenato(X,Y)[1]   <- padre(X,Y).
antenato(X,Y)      <- padre(X,Z), antenato(Z,Y).
```

[1] lettere maiuscole indicano variabili, lettere minuscole (che vedremo in esempi successivi) indicano costanti.

Chiusure transitive nelle basi di dati logiche

Il processo deduttivo per derivare le relazioni virtuali puo' essere condotto con strategie diverse purche' siano garantite due condizioni: la completezza del metodo, cioe' la produzione di tutti gli elementi della relazione, e la sua terminazione. Devono poi essere tenute in conto le caratteristiche di efficienza. Se le clausole che definiscono una relazione virtuale non contengono regole ricorsive, il processo di deduzione non ha problemi (cioe' non e' difficile trovare strategie che garantiscano le condizioni dette sopra). Per esempio, data una domanda si puo' descrivere l'albero di derivazione completo che, non contenendo predicati ricorsivi, ha altezza finita, e "leggere" la frontiera dell'albero come l'espressione di una richiesta dati ad un File System o ad un Sistema di Gestione di Basi di dati.

Invece, il caso in cui intervengano definizioni ricorsive pone dei problemi ed il dibattito sulle corrispondenti strategie di deduzione e' oggi molto vivo. Si vedano, per esempio, [Marq84, Ajel85, Lozi85, Ullm85, Viei85].

1.2. Calcolo del punto fisso

In vari lavori si e' descritta la corrispondenza tra operatori di un linguaggio logico ed operatori dell'algebra relazionale, secondo cui alla relazione virtuale R(X,Y), definita come:

$$(1)\quad R(X,Y) \leftarrow F(X,Y).$$

$$R(X,Y) \leftarrow G(X,Z),\ R(Z,Y).$$

corrisponde la definizione in termini algebrici:

$$(2)\quad R(X,Y) \supseteq F(X,Y).$$

$$R(X,Y) \supseteq G(X,Z)\ |/|^{1,4}_{2,3}\ R(Z,Y).$$

L'espressione $(G(X,Z)\ |/|^{1,4}_{2,3}\ R(Z,Y))$ sta per $\pi_{(X,Y)}\ (G(X,T)\ |/|_{T=S}\ R(S,Y))$ dove π e $|/|$ sono gli operatori proiezione e join dell'algebra relazionale [Ullm82]. Inoltre, nel corso dell'articolo, spesso scriveremo R(X,Y):=(G(X,Z) |/| R(Z,Y)) per indicare l'espressione $R(X,Y):=(G(X,Z)\ |/|^{1,4}_{2,3}\ R(Z,Y))$.

Seguendo [VanE76] e la semantica operazionale di un linguaggio logico li' definita, la relazione R(X,Y) in (1) ha associata la trasformazione T(X,Y) che segue:

$$(3)\quad T(R_i(X,Y)) = F(X,Y)\ \cup\ G(X,Z)\ |/|\ R_i(Z,Y),\ i \geq 0$$

essendo $R_0(X,Y) = \{\ \}$.

Essendo monotoni gli operatori in T e finite le relazioni di base coinvolte, T ha un minimo punto fisso per il piu' piccolo n tale che

$$(4)\quad T(R_n(X,Y)) = T(R_{n+1}(X,Y)).$$

e $R(X,Y) = R_n(X,Y)$.
Il problema e' dunque trovare una strategia che calcoli il punto fisso della trasformazione e che lo faccia in maniera efficiente. Infatti, la computazione del punto fisso secondo la definizione (3) e' in generale molto dispendiosa e spesso inefficiente, nel senso che produce molti fatti non necessari: volendo calcolare ad esempio "antenato(a,Y)" si viene a calcolare il punto fisso della relazione antenato(X,Y) di cui si usano poi soltanto le coppie con primo elemento a.

Molta parte della recente letteratura sulle basi di dati logiche tratta proprio questo problema e concerne proposte di strategie per un calcolo efficiente del punto fisso delle relazioni virtuali ricorsive.

1.3. Organizzazione dell'articolo e contributi

Il secondo paragrafo dell'articolo e' un breve stato dell'arte dove sono considerati lavori apparsi entro la fine del 1985 (riteniamo la precisazione necessaria dal momento che sull'argomento sta convergendo l'attenzione di molti ricercatori ed e' prevedibile che appaiano, nel giro di poco tempo, numerose proposte).
Nel terzo paragrafo dell'articolo ci limitiamo a considerare insiemi di clausole di Horn che definiscono chiusure transitive di relazioni ed esponiamo considerazioni su variazioni algebriche del processo di calcolo del punto fisso nel caso appunto di chiusure transitive. Nel quarto paragrafo le variazioni algebriche del paragrafo precedente sono "riscritte" come procedure tipo Pascal. Si fa poi notare come ciascuna di queste procedure possa essere piu' efficiente nel calcolare la risposta ad una domanda con una particolare istanziazione delle variabili, per calcolare cioe' R(a,Y) o R(X,b) o R(a,b).

2. STRATEGIE EFFICIENTI PER IL CALCOLO DEL PUNTO FISSO

Nel contesto che stiamo considerando, migliorare l'efficienza del calcolo e' migliorare l'interazione con la memoria secondaria, o con il sistema che la gestisce, sia esso File System, Sistema di Gestione di Basi di dati o altro. Si tratta cioe' di migliorare l'interazione tra un sistema di calcolo ed un "sistema esterno" che fornisce argomenti di calcolo al primo. Questa esigenza si ritrova in contesti molto diversi, ad esempio, per rimanere in ambito basi di dati, nei sistemi di gestione di basi di dati distribuite ed ha due linee di intervento (spesso ortogonali) a) la minimizzazione del numero di interazioni tra i due sistemi e b) la minimizzazione del numero di argomenti di calcolo "inutili" nel flusso dati dal sistema esterno al sistema di calcolo.

In questo paragrafo descriviamo brevemente come queste due linee di indagine si ritrovino anche nel caso della valutazione di relazioni definitte ricorsivamente in una base di dati deduttiva.

2.1. Minimizzazione del numero di interazioni con il sistema esterno

Le prime proposte apparse si rifanno da vicino al Prolog, forse meglio sarebbe dire, agli interpreti Prolog disponibili [Kuni82, Vass84]. La loro strategia di calcolo reperisce, su memoria secondaria, un fatto per volta, quindi risulta molto pesante. Infatti, in questo caso ogni ricerca di un nuovo fatto viene a richiedere il colloquio tra l'interprete Prolog ed il File System o il Sistema di Gestione di Basi di Dati perche' quest'ultimo acceda alla memoria secondaria per reperire un dato.

Per una prima ottimizzazione in ambiente Prolog, si e' pensato a diminuire la necessita' di colloquio tra Prolog ed il processo che accede alla memoria secondaria. A questo mirano le strategie compilative [Chan81, Chak82].
Queste strategie prevedono che ad ogni relazione virtuale corrisponda un programma iterativo di calcolo del punto fisso, derivato dalle clausole di Horn che definiscono la relazione, che viene eseguito quando una domanda alla base di dati coinvolge la relazione.

Le strategie compilative derivano il loro nome dal fatto che la definizione di una relazione virtuale espressa in clausole di Horn, sia riscritta per essere eseguita, in un altro linguaggio.

La tecnica di associare un programma di calcolo alle relazioni virtuali puo' essere adottata sia in stretta connessione con il linguaggio Prolog sia in

ambienti dove non si usa Prolog. Si avra' in questo secondo caso un sistema di gestione dati che fornisce ai suoi utenti un linguaggio a clausole di Horn per la descrizione e la gestione della base di dati, ed e' scritto in un qualunque linguaggio, eventualmente Prolog come il sistema descritto in [DeMa85].

2.2. Soltanto dati utili

Un'altra possibilita' di migliorare l'efficienza del calcolo delle relazioni virtuali puo' venire dall'evitare di trovare fatti non utili. A questo mira il lavoro di [Yoko84] dove il calcolo della relazione virtuale R(a,y) e' eseguito calcolando gli elementi R_i in (3) come segue:

$$(5) \quad R_i(a,Y)) \quad = R_{i-1}(a,Y) \cup (R_{i-1}(a,Y) - R_{i-2}(a,Y)) \;|/|_{Y,X}^{1,4}\; F(X,Y)$$

A proposito di questa strategia, in [Viei85], si osserva che applicandola per derivare tutti gli amici di famiglia dell'individuo 'a', essendo la relazione amico-famiglia definita come segue, puo' darsi il caso che se ne trovi soltanto un sottoinsieme.

```
amico-famiglia(X,Y)   <- amico(X,Y).
amico-famiglia(X,Y)   <- padre(X,Z), amico-famiglia(Z,Y).
```

Infatti, il calcolo in (5) e' corretto soltanto se la relazione R e' definita come chiusura transitiva di un'altra relazione (cioe' se in (1) G(X,Y) = F(X,Y)), non e' invece corretto in generale. Piu' precisamente, nel paragrafo che segue faremo vedere che se la relazione R(X,Y) e' una chiusura transitiva ridurre il calcolo del punto fisso al calcolo in (5) permette di trovare il risultato completo e in un numero minore di passi. Mentre in caso contrario questo calcolo non produce una risposta completa.

2.3. Strategie interpretative

Alcuni ricercatori hanno di recente proposto delle tecniche di valutazione che possono definirsi degli interpreti di clausole di Horn (quindi appartenenti alla stessa classe degli attuali interpreti di Prolog) ma tali che, per ogni predicato/relazione valutato trovano contemporaneamente tutti i valori degli argomenti per cui il predicato e' vero, cioe' tutte le tuple della relazione [Lozi85, Ajel85].
Questi interpreti trascrivono le clausole Horn in forma interne a grafo e risultano degli algoritmi di percorrimento di questi grafi con possibile ripetizione di percorso dei cicli.
Riferendoci a Lozinskij, e' interessante notare come venga suggerita una tecnica "generativa" della relazione virtuale: infatti, data una relazione virtuale da valutare, corrispondente ad un nodo della forma interna a grafo, sono identificati i cosiddetti "fatti rilevanti" per la valutazione. Questi sono nodi del grafo corrispondenti a relazioni reali che saranno usate per la valutazione. A questo punto l'algoritmo si comporta come una strategia data-driven (o generativa o bottom-up): le regole che definiscono le relazioni sono applicate "risalendo" verso il nodo corrispondente alla relazione da valutare. Se in questo processo si ha la necessita' di valutare un'altra relazione virtuale (presente in un nodo "and"), il processo di risposta corrente e' sospeso e l'interprete e' chiamato a valutare la risposta alla nuova domanda. Le relazioni definite ricorsivamente sono trattate nello stesso modo: in questo caso l'interprete corrente non "entra in ciclo" bensi' e' sospeso, ed un nuovo interprete ha il compito della valutazione corrispondente ad un percorso del ciclo, con eventuale sua sospensione, nascita di un nuovo interprete e cosi' via fino a che non e' piu' attivato l'interprete che dovrebbe valutare un nuovo "aspetto" della relazione definita ricorsivamente cioe' fino a che, facendo riferimento alla definizione (1), non si hanno tuple soddisfacenti G(X,Z).

3. OTTIMIZZAZIONI DA CONSIDERAZIONI ALGEBRICHE

In questo paragrafo sono raccolte alcune semplici considerazioni algebriche che, come vedremo nel paragrafo seguente, possono essere usate nel contesto di una strategia compilativa, per derivare procedure di valutazione ottimizzata di chiusure transitive.

3.1. Chiusure transitive

Definizione. La chiusura transitiva di una relazione binaria F(X,Y) e' una relazione binaria TC(Z,W) dove (X_i, X_j) e' in TC sse (X_i, X_j) e' in F(X,Y) oppure (X_i, Z) e' in F(X,Y) e (Z, X_j) e' in TC(X,Y).

La definizione e' estendibile a relazioni n-arie interpretando gli elementi X_i, Z, X_j della definizione data come vettori di variabili.

Usando clausole di Horn, la chiusura transitiva della relazione F(X,Y) puo' essere definita come segue:

(6) TC(X,Y) <- F(X,Y).
TC(X,Y) <- F(X,Z), TC(Z,Y).

E' da notare che il calcolo del minimo punto fisso corrispondente a TC(X,Y) puo' in modo equivalente essere effettuato come in (3) oppure come ai punti (7), (8) o (9) derivati da (3):

(7) $TC_{n+1}(X,Y) = U_{i=1}^{n} F(X,Z_1)|/|F(Z_1,Z_2)|/|...F(Z_{i-2},Z_{i-1})|/|F(Z_{i-1},Y)$

$= TC_n(X,Y) \cup F(X,Z_1)|/|...|/|F(Z_{n-1},Z_n)|/|F(Z_n,Y)$

(8) $= TC_n(X,Y) \cup (TC_n(X,Z) - TC_{n-1}(X,Z)) |/| F(Z,Y)$

(9) $= TC_n(X,Y) \cup F(X,Z) |/| (TC_n(Z,Y) - TC_{n-1}(Z,Y))$

Notazioni. Nel seguito useremo le notazioni $TC_i(a,Y)$ e $TC'_i(a,Y)$ per indicare rispettivamente:

$TC_i(a,Y) = \sigma_a{}^2(TC_i(X,Y))$

$= \{(X,Y) \mid (X,Y) \in (F(X,Y) \cup F(X,Z) |/|_{3,4}^{1,2} TC_{i-1}(Z,Y) \text{ and } X=a\}$

$TC'_i(a,Y) = TC'_{i-1}(a,Y) \cup (TC'_{i-1}(a,Y) - TC'_{i-2}(a,Y)) |/|_{3,4}^{1,2} F(X,Y)$

dove $TC'_1(a,Y) = F(a,Y) = \{(X,Y) \mid (X,Y) \in F(X,Y) \text{ and } X=a\}$

Intuitivamente, $TC_i(a,Y)$ e' l'insieme $TC_i(X,Y)$ su cui si opera una selezione per X=a. L'insieme $TC'_i(a,Y)$ e' invece ottenuto col processo di calcolo del punto fisso dove pero' si considera $TC_{i-1}(X,Y)$ ristretto agli elementi con $Y=Z_i$ dove la coppia (X,Z_i) e' in $(TC'_{i-1}(a,Y) - TC'_{i-2}(a,Y))$. $TC'_i(a,Y)$ e' dunque l'insieme calcolato in (5).

Osservazione. TC(a,Y) = TC'(a,Y) (10)

Questo segue da (7) applicando la proprieta' commutativa della selezione con l'unione [Ullm82].

---

[2] σ e' l'operatore di selezione dell'algebra relazionale

Teorema. Consideriamo la trasformazione associata alla relazione TC(X,Y) definita come in (6) e avente minimo punto fisso $TC_n(X,Y)$. Se esiste un intero $k < n$ per cui

$TC_k(a,Y) = TC_{k+1}(a,Y)$
allora $TC_n(a,Y) = TC_k(a,Y)$ (11)

Prova. Supponiamo che $TC_n(a,Y) - TC_k(a,Y) \neq \{\}$. Poiche'
$TC_n(a,Y) - TC_k(a,Y) = \sigma_a(TC_n(X,Y) - TC_k(X,Y)$
vuol dire allora che esiste un elemento (a,Z_t) tale che
$(a,Z_t) \notin \sigma_a(TC_n(X,Y) - TC_k(X,Y))$ ma $(a,Z_t) \notin \sigma_a(TC_k(X,Y))$.
Essendo $TC_n(X,Y) - TC_k(X,Y) = U_{i=k}^{n} F(X,Z_1)|/|F(Z_1,Z_2)|/|...F(Z_{i-1},Y)$, (a,Z_t) e' stato ottenuto congiungendo t coppie
$(a,Z_1), (Z_1,Z_2), ..., (Z_{k-1},Z_k) (Z_k,Z_{k+1}), ..., (Z_{t-1},Z_t)$, $t>k$.
Se t= k+1 si va contro l'ipotesi, se $t > k+1$ per arrivare a comporre (a,Z_t) abbiamo bisogno delle tuple $(Z_k,Z_{k+1}), ..., (Z_{t-1},Z_t)$ che sarebbero elementi di $TC_{k+1}(X,Y), ..., TC_{t-1}(X,Y)$ ancora contro l'ipotesi.

Il caso descritto in [Viei85] serve come controesempio all'analogo teorema nel caso di una relazione che non sia chiusura transitiva.

Lemma. $TC'_k(a,Y) = TC_n(a,Y)$ (12)

Segue da (11) e da (10).

4. SCHEMI DI PROCEDURE DI VALUTAZIONE

In questo paragrafo spieghiamo che interesse abbiano le considerazioni algebriche viste nei paragrafi precedenti. Per prima cosa, gli schemi di calcolo ai punti (8) e (9) sono "riscritti" in una forma procedurale alla Pascal. Si fa poi notare come ciascuna procedura possa essere piu' efficiente nel calcolare la risposta ad una domanda con una particolare istanziazione delle variabili, per calcolare cioe' TC(a,Y) o TC(X,b) o TC(a,b).

4.1. Schemi di procedure

Dallo schema di calcolo al punto (8) si deriva la procedura alla Pascal che segue.

```
procedure TC;
    TC(X,Y) := F(X,Y);
    △(X,Y):= F(X,Y);
    while △(X,Y) ≠ {} do
        begin
            △(X,Y) := △(X,Y) |/| F(X,Y)
            TC(X,Y) := TC(X,Y) ∪ △(X,Y)
            if cyclic then △(X,Y) := △(X,Y) - TC(X,Y)
        end
end.
```

In questa procedura TC(X,Y) e △(X,Y) sono "insiemi di tuple" che possono stare tanto in memoria secondaria, per esempio come nuove relazioni create nella base di dati, o essere mantenuti in memoria centrale. Questo aspetto va deciso tenendo conto della dimensione di TC(X,Y) e △(X,Y) e, quindi, di F(X,Y), di △(X,Y) |/| F(X,Y), TC(X,Y) ∪ △(X,Y). Facciamo anche l'ipotesi che gli operatori dell'algebra relazionale siano estesi ad operare su questi insiemi di tuple. L'operazione di assegnazione := e' anch'essa da considerarsi estesa a operare in memoria centrale oppure secondaria, dove corrispondera' all'operazione di "copy" di files o relazioni. Nell'istruzione " if cyclic then △(X,Y) := △(X,Y) - TC(X,Y) " la variabile booleana " cyclic" e' true quando la relazione F(X,Y) e'

riflessiva, false quando non lo e'.

Usando il medesimo punto (8) ed il risultato in (12), si puo' invece derivare la procedura:

```
procedure TC_x;
     TC(Y) := π2 (F(a,Y));
     Δ(Y):= π2 (F(a,Y));
     while Δ(Y) ≠ {} do
          begin
               Δ(Y) := Δ(Y) |/|3^1,2 F(X,Y)

               TC(Y) := TC(Y) U Δ(Y)
               if cyclic then Δ(Y) := Δ(Y) - TC(Y)
          end
end.
```

Ancora il risultato in (12) applicato alla variazione di calcolo del punto fisso in (9) ci porta a derivare la procedura che segue.

```
procedure TC_y;
     TC(X) := π1 (F(X,b));
     Δ(X):= π1 (F(X,b));
     while Δ(X) ≠ {} do
          begin
               Δ(X) := F(X,Y) |/|2,3^1 Δ(X)

               TC(X) := TC(X) U Δ(X)
               if cyclic then Δ(X) := Δ(X) - TC(X)
          end
end.
```

Le procedure TC_x e TC_y sono candidate a poter essere scelte come piu' efficaci, dal punto di vista della prestazione, procedure di calcolo di risposte a domande rispettivamente del tipo TC(a,Y) e TC(X,b) invece di calcolare TC(X,Y) e poi operare la selezione voluta. Nel caso invece di una domanda del tipo TC(a,b) in cui si vuole conoscere se gli elementi a e b sono tra loro nella relazione TC, si deve considerare la selettivita' degli attributi X ed Y della relazione F(X,Y). Nel caso in cui Y abbia maggiore selettivita', potra' essere una efficace procedura di calcolo la seguente:

```
procedure TC_xy;
     TC(X) := π2 (F(X,b));
     Δ(X):= π2 (F(X,b));
     while (Δ(X) ≠ {}) and ('a' ∉ Δ(X)) do
          begin
               Δ(X) := F(X,Y) |/|2,3^1 Δ(X)

               TC(X) := TC(X) U Δ(X)
               if cyclic then Δ(X) := Δ(X) - TC(X)
          end
end.
```

E' da notare che quando si dovessero valutare le relazioni risposta alle domande TC(a,Y), TC(X,b), o TC(a,b) e F(a,Y) o F(X,b) non sono presenti nella base di dati, le procedure viste terminano alla prima verifica che Δ(X) sia ≠ {} o Δ(Y) sia ≠ {}, cioe' dopo un unico accesso alle relazioni reali che ha determinato la non esistenza di tuple F(a,Y), F(X,b).

5. CONCLUSIONI

In questo paragrafo conclusivo accenniamo agli obiettivi della nostra attuale ricerca sulle basi di dati logiche, che possono riassumersi come segue.
Un primo argomento di ricerca e' una analisi della complessita' delle definizioni ricorsive in una base di dati ed una verifica della "ampiezza" della classe di relazioni che possono essere definite come chiusure transitive, da una forma che inizialmente non lo e'. Infatti, nel caso di chiusure transitive le procedure di risposta viste nel paragrafo precedente sono molto semplici ed efficienti. Un primo risultato di questa analisi e' la determinazione di una classe di definizioni ricorsive che non sono chiusure transitive ma che possono essere riscritte come tali per mezzo della seguente trasformazione sintattica:

una definizione della relazione p(X,Y) del tipo:

```
(1)   p(X,Y)   <- q1(X,Y,U1), ..., qn(X,Y,Un)
      p(X,Y)   <- r1(X,Z,V1), ..., rm(X,Z,Vm), p(Z,Y)
```

puo' essere riscritta come:

```
(2)   p(X,Y)      <- q(X,Y)
      p(X,Y)      <- ct(X,Z,Y), q(Z,Y)
      ct(X,Z,Y)   <- r(X,Z,Y)
      ct(X,Z,Y)   <- r(X,Z1,Y), ct(Z1,Z,Y)
      q(X,Y)      <- q1(X,Y,U1), ..., qn(X,Y,Un)
      r(X,Y)      <- r1(X,Y,V1), ..., rm(X,Y,Vm)
```

In [Demo85] si fa vedere come i due insiemi di clausole (1) e (2) siano "computazionalmente equivalenti" (mentre non sono "logicamente equivalenti").

Un'altra direzione di ricerca e' la definizione di un sistema di gestione di basi di dati deduttive dove viene adottato un approccio di tipo compilativo. Riteniamo infatti che la compilazione renda piu' facile l'utilizzo delle tecniche di ottimizzazione dei sistemi di gestione di basi di dati tradizionali ed insieme permetta piu' ampie ed originali ottimizzazioni. Infatti analizzando la "visione globale" della procedura di valutazione di una relazione fornita dal processo di compilazione, si possono per esempio definire sulle relazioni reali (o eventualmente su quelle risultato intermedio della computazione) nuove strutture di accesso, per esempio indici, poi eliminati alla fine de processo oppure potrebbero essere definite nuove tecniche di ottimizzazione che coinvolgono piu' operazioni sulla base di dati come si e' cominciato a fare in [Sell85].

Da ultimo citiamo un lavoro di analisi di varie proposte cui e' legata la definizione di un paradigma per il loro confronto. In un momento in cui appaiono numerosissime nuove strategie ci sembra infatti indispensabile un'attivita' che "trascriva" in un linguaggio comune le varie proposte e permetta di capirne il "magico" funzionamento [Banc85].

RINGRAZIAMENTI. Le persone che hanno contribuito, favorendo il mio interesse per l'argomento o con chiarimenti, sono molte. Tra queste un grazie particolare va a R. Demolombe, P. Giolito, E. Giovannetti, F. Honsell, G. Lolli, M. Tilli per le numerose ed attente discussioni.

BIBLIOGRAFIA

[Ajel85] L. Ajello, C. Cecchi, Adding a closure operator to the extended relational algebra: a further step towards the integration of data base techniques and logic programming, Rapporto Interno, Dipartimento di

Informatica e Sistemistica, Univ. Roma, Settembre 1985.

[Banc85] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman, Magic sets and other stranger ways to implement logic programs, MCC Technical Report Number: DB-121-85, 1985.

[Chan78] C. L. Chang, DEDUCE 2: Further Investigations of Deduction in Relational Databases, in Logic and Databases, Gallaire and Minker (eds.), Plenum Publishing Corp., 1978.

[Chan81] C. L. Chang, On Evaluation of Queries containing Derived Relations in a Relational Database, in Advances in Database Theory, Gallaire, Minker, Nicolas (eds.), Plenum Publishing Corp., 1981.

[Chak82] Chakravarti, J. Minker, D. Tran, Interfacing Predicate Logic Languages and Relational Databases, Atti First Intern. Conf. on Logic Programming, Marseille, 1982.

[DeMa85] B. Demo, M. P. Manzoni, Architettura di un sistema di gestione di basi di dati deduttive, Rapporto Interno del Dip. di Informatica, Univ. di Torino, Ottobre 1985.

[Demo85] B. Demo, G. Lolli, M. Tilli, On the reducibility of a class of Horn clauses to transitive closure, Rapporto Interno del Dip. Informatica Univ. di Torino, Luglio 1985.

[Gall83] H. Gallaire, Logic Databases vs Deductive Databases, Atti Logic Programming Workshop 1983, Albufeira, 1983.

[Kuni82] S. Kunifuji, H. Yokota, Prolog and relational data bases for Fifth Generation Computer Systems, Atti del Workshop "Logical bases for Databases, Tolosa, Dicembre 1982.

[Lozi85] E. L. Lozinskii, On bottom-up Query Evaluation in Deductive Databases, Atti IJCAI 85, Los Angeles, 1985.

[Marq84] G. Marque-Pucheau, Algebraic structure of answers in a recursive logic database, Rapporto Interno Ecole Normale Superieure, Paris, 1984.

[Sell85] T. K. Sellis, L. Shapiro, Optimization of extended database query languages, Atti SIGMOD 85, Austin, 1985.

[Ullm85] J. Ullman, Implementation of logical Query languages for Databases, Atti SIGMOD 85, Austin, 1985.

[Vass84] Y. Vassiliou, J. Clifford, M. Jarke, Access to specific declarative knowledge by Expert Systems: the impact of logic programming, Decision Support Systems, Vol. 1, 1, 1984.

[Viei85] L. Vieille, Le traitement des axiomes recursifs dans les bases de donnees deductives, Atti Seminaire de programmation logique, CNET, Tregastel, 1985.

[Yoko84] Yokota, Kunifuji, et al., An enhanced Inference Mechanism for generating Relational Algebra Queries, Att. SIGMOD 84, Boston, 1984.

INTEGRAZIONE DI AMBIENTI GRAFICI E DATABASE LOGICI

ASIRELLI P. (+), CASTORINA P. (*), MAINETTO G. (-)

(+) Istituto di Elab. dell'Informazione. - CNR- PISA
(*) Kinetics Technology International - ROMA
(-) CNUCE - CNR- PISA

ABSTRACT

Il lavoro presenta una proposta di integrazione tra grafica e basi di dati logiche che rappresenta il punto di avvio per la costruzione di un ambiente di sviluppo di applicazioni grafiche CAD/CAM particolarmente potente e flessibile poiche' permette di combinare le facility messe a disposizione dalle piu' recenti proposte di standard grafici, GKS-3D e PHIGS, con quelle tipiche degli ambienti per la programmazione logica. L'integrazione proposta consente di definire in modo uniforme oggetti grafici e non grafici, di costruire relazioni fra di essi, di dedurre proprieta' degli oggetti grafici e non, ed inoltre offre la possibilita' di effettuare controlli di consistenza rispetto a vincoli di integrita' asseriti sugli oggetti.

1. Introduzione

La possibilita' di realizzare, mediante l'uso di linguaggi basati su clausole Horn, basi di dati deduttive, e' attualmente un tema di grande interesse per la comunita' scientifica, anche Europea (numerosi sono i progetti ESPRIT esistenti sul tema della integrazione della Programmazione Logica e Basi di Dati [Asirelli 85c]). Cio' e' dovuto alle possibilita' che si intravedono di potenziare, con uno strumento logico, basi di dati tradizionali esistenti. Assume quindi importanza, non solo la integrazione del Prolog, con linguaggi di query di basi di dati disponibili, ma anche la integrazione di basi di dati logiche, cioe' implementate e gestite con linguaggi logici, con l'altro tipo di basi di dati, per una loro ridefinizione o anche per la definizione di views diverse della base di dati esistente.

La rilevanza dell'uso dei linguaggi logici nel campo delle basi di dati, a parte gli aspetti dichiarativi di tali linguaggi, e' dovuta, in molta parte, alla uniformita' del linguaggio con cui si possono esprimere sia i fatti che le regole (aspetto estensionale ed intensionale) [Gallaire 84, Nicolas 82], sia i vincoli di integrita' che le transazioni (operazioni di modifica allo stato del database)[Mauro 85], ma anche dalla possibilita' di utilizzare un unico strumento, l'interprete, da utilizzare per query, per la dimostrazione della consistenza rispetto ai vincoli d'integrita' [Nicolas 82 Asirelli 84, Asirelli 85d] e la esecuzione di operazioni di modifica. Si deve invece notare che, fino ad ora, tutti questi aspetti venivano rappresentati con formalismi diversi e realizzati con meccanismi ad hoc.

D'altra parte, la possibilita' di fare deduzione ed effettuare controlli sulle proprieta' e' uno strumento molto potente da utilizzare negli ambienti di sviluppo in generale. In particolare, per quanto concerne questo lavoro ci siamo orientati alla definizione di un linguaggio e alla progettazione di meccanismi che permettano lo sviluppo di applicazioni CAD/CAM, dove e' evidente il vantaggio di fornire un ambiente di programmazione che permetta non solo la costruzione (descrizione di) oggetti componendoli tra loro, ma anche la loro realizzazione facendo uso di strumenti che ne guidino e controllino la correttezza rispetto a certi vincoli essenziali per la progettazione.

Ci siamo quindi proposti di investigare l'impiego dei data base deduttivi logici nel campo della grafica per applicazioni di tipo CAD/CAM. L'obiettivo e' quello di trovare un giusto compromesso tra basi di dati logiche e sistemi grafici per ottenere un ambiente di sviluppo di applicazioni CAD/CAM che combini i vantaggi di entrambi [Swinson 80, Camacho 84, Asirelli 85a, Asirelli 85b].

A questo proposito, e' in atto una collaborazione tra la Kinetics Technology International, l'Istituto di Elaborazione dell'Informazione ed il CNUCE per la la definizione e la realizzazione di un sistema di gestione per una base di dati logica, integrato con un linguaggio grafico (Graphic-EDBLOG).

E' stato preso in considerazione e viene proposto un approccio alla integrazione che permette di ridefinire in termini logici tutto l'ambiente grafico, lasciando alle apparecchiature grafiche le sole operazioni di I/O. Un esempio viene presentato in appendice.

2. EDBLOG

Una base di dati deduttiva, che e' definita come un insieme di fatti (componente *estensionale*) e un insieme di regole (componente *intensionale*), puo' essere vista come una teoria del prim'ordine. Data la interpretazione procedurale delle clausole Horn [Kowalski 74] e posto che i fatti e le regole della base di dati siano espressi con clausole Horn, possiamo considerare una base di dati come un programma logico [Gallaire 84, De Santis 85, Asirelli 85c]. E' dunque possibile trattare i problemi relativi alle basi di dati come problemi dei programmi logici in generale.

Sostanzialmente, si e' partiti considerando un DBMS, DBLOG [Asirelli 84, De Santis 85, Asirelli 85c], in cui la base di dati e' un programma logico piu' un insieme di formule che esprimono i vincoli di integrita' di due tipi:*Vincoli d'Integrita'* e *Controlli* .

Queste formule devono risultare vere nel modello minimo del programma logico [Lloyd 84], in modo da garantire che la base di dati sia corretta rispetto ai vincoli di integrita'. In particolare, le formule dei vincoli d'integrita' vengono utilizzate per modificare il programma logico, in modo che l'insieme dei fatti da esso derivabili siano solo quelli che soddisfano tali vincoli (il modello minimo e' anche modello dei vincoli d'integrita'); le formule dei controlli invece, vengono utilizzate periodicamente, per controllare che le modifiche alla base di dati preservano la consistenza rispetto a questo tipo di vincoli.

Oltre al controllo dei vincoli DBLOG provvede a gestire la base di dati mediante le operazioni elementari di modifica (aggiunta e cancellazione di fatti, regole, vincoli e controlli) e di interrogazione.

Alcune estensioni a DBLOG si sono rese necessarie e possono brevemente essere riassunte come segue:

a) ammissibilita' di regole deduttive ricorsive tali da garantire la completezza delle query, anche se in forme particolari [Barbuti 85]. Questo per rilassare il vincolo gerarchico richiesto da DBLOG, che risulta essere un vincolo estremamente restrittivo;
b) possibilita' di definire transazioni, cioe' operazioni sul DB definite come sequenza di altre transazioni od operazioni elementari che ammettano momentanee inconsistenze del DB;
c) verifica dei controlli al termine della esecuzione di un goal, anziche' su richiesta dell'utente.

La versione estesa (EDBLOG) [Mauro 85] include l'interprete per il linguaggio delle transazioni, basato anch'esso sulla logica, che permette di condizionare la esecuzione delle transazioni a pre-condizioni (controlli specifici della transazione in oggetto) e post-condizioni (controlli relativi alla situazione del DB dopo la esecuzione della transazione). L'uso di pre e post condizioni nella definizione delle transazioni, consente di separare i controlli globali del database da quelli relativi a particolari operazioni, riducendo cosi' il numero di controlli di consistenza che devono essere fatti periodicamente.

Riassumendo, EDBLOG risulta cosi' composto:

1) teoria delle clausole Horn, costituita da:

1.1 *un insieme di fatti* , in forma di clausole Horn unitarie (Componente Estensionale del DB) ;
1.2 *un insieme di regole*, in forma di clausole Horn definite non unitarie (Componente Intensionale del DB);

2) *un insieme di vincoli di integrita'* (integrity constraints o IC) che sono formule del tipo:

$$A \rightarrow B_1,\ldots,B_n .$$

la cui interpretazione informale e' che ogni volta che **A** e' vera allora B_1 e . ...e B_n, devono essere veri;

3) *un insieme di controlli*, che sono formule del tipo 2) oppure :

$$A_1 \wedge \ldots \wedge A_m \rightarrow B_1,\ldots,B_n$$

$$\rightarrow B_1,\ldots,B_n$$

$$A_1 \wedge \ldots \wedge A_m \rightarrow$$

4) teoria che definisce le operazioni elementari di modifica a 1) mediante le operazioni:

add (A)
remove(A)

dove **A** e' un fatto od una regola di 1).

5) insieme di transazioni definte come:

$$A \leftarrow Pr_1,\ldots, Pr_i \mid B_1,\ldots,B_m \mid Ps_1,\ldots, Ps_n$$

4) e 5) possono anche essere considerate amalgamate [Bowen 82], in cui i vari **Pr** e **Ps** sono precondizioni e postcondizioni da verificare in 1) (goal da eseguire), ed il cui corpo $B_1,\ldots,B_m$ e' costituito da operazioni elementari di modifica a 1) e/o da altre transazioni definite sempre in 5). L'interpretazione di un goal e' fatta nella teoria 5) (4 e 5 amalgamate) e prosegue in 1) o 4) (4 e 5 amalgamate) a seconda che si debbano valutare pre/postcondizioni di una transazione e/o operazioni elementari.

3. Grafica

Per quanto concerne la parte grafica dell'ambiente integrato che vogliamo realizzare, abbiamo preso in considerazione gli standard grafici GKS-3D [ISO 83, ISO 84] e PHIGS [ANSI 84]. Da un'analisi comparativa dei due standards [Biagi 85a] risulta che il piu' avanzato ambiente grafico attualmente proposto e' quello del PHIGS. In particolare ci siamo riferiti ad una piu' recente proposta, PHOGS [Biagi 85b], un sottoinsieme del PHIGS, particolarmente adatto a microcomputers senza hardware specializzato per la grafica.

La evoluzione degli standard grafici sembra portare alla definizione di un modello in cui gli oggetti grafici (segmenti) sono definiti in modo *relativo*. Gli oggetti sono costituiti da una descrizione che e' di fatto data da una sequenza di comandi grafici che vengono interpretati a tempo di visualizzazione. Quindi, l'associazione delle proprieta' e delle trasformazioni avviene a tempo di *visualizzazione*, cioe' gli oggetti hanno valori assoluti per la posizione, colore e per le altre proprieta' grafiche, solo al momento in cui vengono visualizzati.

Si puo' in sostanza riassumere la tendenza della grafica come:

dipendenza dall'Hardware	**verso**	dipendenza dagli standards
descrizione assoluta deli oggetti	**verso**	descrizione relativa (modello)
compilazione	**verso**	interpretazione

L'ultimo punto e' piuttosto importante, dato che il *traversing* (cioe' l'interpretazione dei comandi grafici di un oggetto) potrebbe essere molto frequente in programmi applicativi e comportare un notevole appesantimento del tempo di proiezione.

L'altro aspetto rilevante, rispetto alle tendenze precedenti, e' che le descrizioni relative degli oggetti, permettono di organizzare in modo gerarchico gli oggetti stessi, ed acquista significato l'opportunita' di editare segmenti (cambio dei comandi grafici che lo definiscono).

Le attivita' principali di un programma applicativo grafico sono: *creazione, cancellazione, inquiry, editing* e *visualizzazione* di segmenti. Queste operazioni, esclusa quella di visualizzazione, implicano l'esistenza e la gestione di una base di dati di segmenti, che normalmente viene effettuata dagli ambienti grafici.

4.Integrazione

Ci siamo proposti di specificare, usando il formalismo delle clausole Horn, un modello di standard grafico ad oggetti in cui le operazioni grafiche siano il piu' possibile espresse in termini logici e rimandando alla parte grafica solo le funzioni di input ed output sulle apparecchiature in oggetto.

La descrizione dei segmenti puo' essere vista come un DB che include anche informazioni relative alle relazioni gerarchiche tra i segmenti stessi espressi mediante il comando *'execute* (identificatore_di_segmento)'.

La *visualizzazione* lancia la interpretazione della descrizione di un segmento come insiemi di comandi grafici, per produrre una immagine sul dispositivo di output. La visualizzazione dei sottosegmenti procede invece per interpretazione dei relativi comandi *execute*.

Consideriamo i segmenti espressi mediante *fatti* (clausole unitarie) del tipo:

seg (a,(polyline (p1,p2,p3,p4), set_color(rosso), execute(g))).

Le operazioni sul DB dei segmenti sono:

- *creazione* di segmenti;
- *cancellazione* di segmenti;
- *modifica* della descrizione di un segmento;
- *query* al data base di segmenti;
- *visualizzazione* dei segmenti sul video utilizzando un *ambiente* di visualizzazione.

Ovviamente, la formalizzazione in termini di clausole Horn, puo' essere direttamente eseguibile in Prolog. Le primitive di tale linguaggio devono essere estese in modo da inglobare le operazioni grafiche di input e output, per esempio: *polyline*, *set_color* ecc.. In EDBLOG la *visualizzazione* e la *execute* costitutiscono parte della teoria delle transazioni e possono essere definite mediante pure clausole Horn, oppure utilizzando il formalismo con precondizioni, come segue:

visualizzazione (X) ← *seg* (X, Desc) | *set_env* , Desc.

execute (X) ← *seg* (X, Desc) | *save_env*, Desc, *restore_env*.

In cui *seg* (X, Desc) serve da precondizione alle operazioni successive, e in questo caso serve anche da retrieve dal database di segmenti; la | sta per una operazione di commitment a quella clausola; il fallimento della parte a sinistra di | causa fallimento della relativa operazione (*visualizzazione* o *execute*).

save_env e *restore_env* sono comandi grafici (primitive del linguaggio logico) che permettono di salvare l'ambiente attuale e di ripristinarlo dopo la visualizzazione del 'sottosegmento'. Questa caratteristica riflette l'approccio degli standard grafici considerati (PHIGS e PHOGS), in cui le modifiche locali ai sottosegmenti non influenzano l'ambiente dei 'supersegmenti'.

Per le operazioni primitive il sistema grafico deve prevedere una comunicazione di fallimento della operazione qualora questa non vada a buon fine (p.e. errore nei dati), in modo da poter scatenare il backtracking. Inoltre, per tali operazioni (tra cui *save_env* e *restore_env*) deve esistere una operazione di "ripristino" della situazione in caso *backtracking*.

A tale proposito, oltre alla *save_env* e *restore_env* si possono distinguere le operazioni primitive il cui effetto e' puramente grafico (p.e. tracciare una spezzata), da quelle che modificano l'ambiente (p.e. settaggio del colore).Mentre per le prime e' sempre necessaria una operazione di

undo_op , per le seconde cio' e' necessario solo se si ammette che nella descrizione di un segmento ci siano anche formule atomiche qualsiasi, cioe' formule atomiche su altri predicati definiti allo stesso livello dei segmenti o delle operazioni non primitive. Ad esempio:

seg (a,(set_color (bianco),pred(X,Y),set_color(X), polyline (p1,Y,p3,p4))).

Supponendo di visualizzare un segmento di cui a e' un sottosegmento, un fallimento di 'polyline(p1,Y,p3,p4)' richiede un *undo* di 'set_color(X)' e la ricerca di nuovi valori per X e Y da parte di 'pred(X,Y)'. Mentre per :

seg (a,(set_color (bianco), polyline (p1,p2,p3,p4))).

un errore in polyline non richiederebbe anche un undo di set_color(bianco), perche', per come e' definita la *execute* , basterebbe una propagazione all'indietro del fallimento ed il corretto *undo* della *save_env*.

A tale proposito, mentre per la *save_env* il suo corrispondente undo e' *restore_env*, per quest'ultima e' necessario conoscere l'ambiente locale precedente per poterlo ripristinare in fase di backtracking, e questa e' una operazione che deve essere gestita dal sistema grafico. Il sistema grafico deve inoltre prevedere che l'undo di operazioni non andate a buon fine debba poter operare su qualunque situazione lasciata sospesa, ad esempio mantenedo un 'transition_log', in modo da poter fare l'undo delle singole operazioni primitive (del sistema grafico).

Quando, per ogni operazione primitiva, il sistema grafico fornisce sia il fallimento (per terminazione anomala) che le relative operazioni di undo, la gestione del backtracking di tali operazioni puo' essere fatta mediante un programma Prolog di interfaccia con la grafica come segue:

polyline (X,Y,Z,W) ← polyline_g(X,Y,Z,W).

polyline (X,Y,Z,W) ← undo_polyline (X,Y,Z,W), **fail**.

in cui polyline_g e' la primitiva grafica e polyline la sua implementazione in Prolog.

L'input dei segmenti puo' essere pensato direttamente come effetto di una transazione:

create_segment(X,Desc) ← *add* (seg(X,Desc)).

Questa operazione deve essere lanciata al termine di una serie di transazioni che, sulla base della applicazione definita e mediante interazione con l'utente della stessa, raccolgono tutte le informazioni grafiche da inserire in Desc.

Le caratteristiche principali di questo approccio sono:

- una definizione piu' formalizzata dell'ambiente grafico;

- una visione uniforme di aspetti grafici e non grafici, che risulta particolarmente utile per applicazioni di CAD/CAM;

- il concetto di segmento risulta esteso per la possibile presenza nella definizione di operazioni di *inquiry* sull'ambiente. Per esempio:

seg (b, (*inquiry* (color (blue)), *execute* (g))).

per cui, in fase di visualizzazione del segmento 'b', il segmento 'g' verra' visualizzato **solo se**

l'ambiente avra' 'blue' come colore attuale. In questo modo la visualizzazione dipende dalle proprieta' attuali dell'ambiente;

- la definizione del segmento puo' includere anche proprieta' logiche;

- e' possibile definire relazioni o proprieta' di un segmento, creando fatti e regole che non fanno parte della definizione del segmento;

- la possibilita' di esprimere e verificare vincoli sui segmenti e sulle loro operazioni sia grafiche che non (vincoli di integrita', controlli, pre e postcondizioni nelle transazioni).

Si deve notare che in questo approccio l'insieme dei fatti e' considerato essere espresso in clausole Horn. Si puo' pero' prevedere che l'insieme dei fatti grafici (segmenti) sia mantenuto in un data base grafico e che il recupero delle informazioni rappresentanti il segmento avvenga mediante query a tale database. Questo ulteriore livello di integrazione non cambierebbe la filosofia del sistema risultante. Il controllo delle operazioni e quindi anche le inserzioni di nuovi segmenti o modifiche ad essi, passerebbe ancora attraverso EDBLOG. Tali operazioni elementari sui fatti grafici verrebbero convertite nella relativa operazione del database grafico sottostante.

In questo modo si andrebbe incontro a quelle esigenze di efficienza di gestione delle informazioni che alla base di varie proposte che vanno verso la integrazione della programmazione logica con basi di dati esistenti (progetti ESPRIT EPSILON, ecc...).

Un esempio di questo approccio viene dato in Appendice.

Attualmente il sistema presentato e' in fase di sviluppo su di un micro computer che supporta il sistema operativo UNIX System V, con UNSW Prolog, compatibile con DEC 10 Prolog.

5. Conclusioni

E' stato presentatio un possibile approccio alla integrazione della grafica con un sistema di gestione di basi di dati logiche, EDBLOG in cui le operazioni grafiche (di basso livello) vengono considerate come primitive del linguaggio logico (Prolog), utilizzato per l'implementazione del DB. In questo modo, l'interpretazione di clausole contenenti predicati grafici, comporta l'esecuzione del corrispondente comando grafico; l'ambiente grafico "standard" verrebbe ridefinito in EDBLOG.

Questo tipo di integrazione [Asirelli 85a, Asirelli 85b], considera i segmenti come oggetti del DB logico, sui quali possono essere definite sia relazioni grafiche che non grafiche. Analogamente si possono definire transazioni i cui effetti possono essere grafici o non grafici. I vantaggi di questo approccio si hanno dal punto di vista della omogeneita' di trattamento delle informazioni (grafiche e non), e anche delle operazioni di modifica allo stato degli oggetti. L'ambiente grafico originario puo' essere completamente ridefinito nell'ambiente logico. Cio' rende estremamente chiaro il significato delle operazioni grafiche (stile dichiarativo rispetto a programmazione Assembler o Fortran), e permette anche di estendere le potenzialita' dell'ambiente grafico originario, ad esempio, definendo altre relazioni (vedi appendice).

6. Appendice

Supponiamo che la nostra base di dati sia costituita dai seguenti **fatti** relativi ad oggetti grafici:

seg (a, (polyline (a1,a2,a3,a4))).
seg (b, (set_color (blue),execute(a))).

e da altri fatti rappresentanti proprieta' non grafiche:

prezzo (a,1000).
prezzo (b,10000).

Le **regole** ci permettono di esprimere alcune proprieta' degli oggetti grafici:

sotto_seg (Seg1,Seg2) ← *seg* (Seg1 ,Desc), *member* (Desc,execute(Seg2)).
complanar (Seg1,Seg2) ← *is_complanar* (Seg1, Plan), *is_complanar* (Seg2, Plan).
is_complanar (Seg, Plan) ← *seg* (Seg, Desc), *onplan* (Desc, Plan).
onplan .....

Si possono anche esprimere **vincoli :**

prezzo (X,Y) → Y > 100.
.....

e **controlli :**

seg (X, Desc), *sotto_seg* (X,Y) → *seg* (Y,Desc).
sono_complanari (X, Y) →*complanar* (X,Y).
.....

La creazione di un nuovo segmento puo' essere condizionata dalla eventuale presenza di un segmento con lo stesso nome se la transazione ha la seguente definizione:

create_seg (X,Desc)← *not* (seg(X,Y)) | *add* (seg(X,Desc)).

mentre la modifica (editing) ha senso solo se e' presente un segmento con lo stesso nome:

edit_seg (X,Desc)← seg(X,Y) | *remove* (seg(X,Y)), *add* (seg(X,Desc)).

La complanarita' tra due segmenti puo' essere asserita mediante la seguente transazione:

assert_complan (X,Y) ← *commplanar* (X,Y) | *add* (sono_complanari(X,Y)).

La terminazione con successo di un goal come:

← *assert_seg* (c, (*execute* (f))).

provoca la verifica dei controlli che in questo caso fallisce perche' il segmento f non e' stato ancora definito (violazione del primo controllo).

7. Bibliografia

[ANSI 84] **ANSI X3H3**, "PHIGS Functional description", 1984.

[Asirelli 84] **P. Asirelli, M. Martelli**, "Integrity Constraints, Redundancy and Consistency in Logic Data Bases", CNUCE Int. Rep. C84-24, 1984.

[Asirelli 85a] **P. Asirelli, P. Castorina, G. Mainetto,** "Programmazione Logica, Basi di Dati Logiche e Grafica", *AICOGRAPHICS'85*, Milano, 4-8 Novembre, 1985.

[Asirelli 85b] **P. Asirelli, P. Castorina, G. Mainetto**, "Logic Databases and Graphics: A proposal for Integration", I.E.I. Int. Rep. B85-10, Settembre,1985.

[Asirelli 85c] **P. Asirelli, M. De Santis, P. Franceschi, C. Simonelli, G. Levi, M. Martelli,** "The Knowledge Base Approach in thwee Epsilon Project", *ESPRIT Technical Week 1985* , Bruxelles, 23-25 Settembre, 1985.

[Asirelli 85d] **P. Asirelli, M. De Santis, M. Martelli**, "Integrity Constraints in Logic Data Bases", accettato per pubblicazione su *Journal of Logic Programming*, 1985.

[Barbuti 85] **R. Barbuti, M. Martelli**, "Programming in a Generally Functional Style to Design Logic Data Bases", Sottoposto per pubblicazione.

[Biagi 85a] **B. Biagi, C. Montani, R. Scopigno**, "Analisi comparativa di standard e prodotti grafici", I.E.I. Int. Rep., Maggio, 1985.

[Biagi 85b] **B. Biagi, C. Montani, R. Scopigno**, "PHOGS (PHIGS Oriented Graphic System): Proposal For an Interactive Graphic Language. First Part: Basic Concepts and Functionalities", I.E.I. Int. Rep., Giugno, 1985.

[Bowen 82] **K.A. Bowen, R. A. Kowalski**, "Amalgamating langauage and metalanguage in logic programming", *Logic programming*, eds K.L. Clark & S.A. Tarnlund, N. Y.,1982.

[Camacho 84] **J. Camacho Gonzalez, M.H. Williams, I. E. Aitchison**, "Evaluation of the Effectiveness of Prolog for a CAD Application", IEEE CG & A, N. 2, 1984.

[De Santis 85] **M. De Santis**, "Logic Programming e Databases: un Ambiente di Sviluppo adatto alla Gestione dei Vincoli di Integrità", Tesi di Laurea, Dip. Informatica, Universita' di Pisa, 1985.

[Gallaire 84] **H. Gallaire, J. Minker, J. Nicolas**, "Logic and Databases: a deductive approach",*Computing Surveys*, N. 2, 1984.

[ISO 83] **ISO Graphical Kernel System**. *DIS 7942*. 1983.

[ISO 84] **ISO TC97/SC5/WG2**, "Extensions of GKS to 3D", Working Draft, N277, 1984.

[Kowalski 74] **R. A. Kowalski**, "Predicate Logic as a Programming Language", *Proceedings IFIP 74*, 1974.

[Lloyd 84] **J. Lloyd**, Foundation of Logic Programming, Springer Verlag, New York, 1984.

[Mauro 85] **F. Mauro**, "Basi di Dati Logiche: un approccio al trattamento delle Transazioni", Tesi di Laurea, Dip. Informatica, Universita' di Pisa, Dicembre 1985.

[Nicolas 82] **J. Nicolas**, "Logic for Improving Integrity Checking in Relational Data Bases", *Acta Informatica*, 18, 1982.

[Robinson 65] **J.A. Robinson**, "A machine-oriented Logic Based on the Resolution Principle", *JACM*, N. 12,1965.

[Shapiro 83] **E. Y. Shapiro, A. Takeuchi**, "Object-Oriented Programming in Concurrent Prolog",*New Generation Computing*, N. 1, 1983.

[Swinson 80] **P.S.G. Swinson**, Prescriptive to Descriptive Programming: A Way Ahead for CAAD, *Proc. of the Logic Programming Workshop*, Debrecen, Hungary, 1980.

PROGETTO MICROPROLOG: IL PROLOG NELLA DIDATTICA

R.M. Bottino - P. Forcheri - M.T. Molfino
Istituto per la Matematica del C.N.R.
Via L.B. Alberti 4 16132 GENOVA

ABSTRACT

Il linguaggio Prolog e' oggi di notevole interesse nel campo delle applicazioni del calcolatore nella didattica, come confermano le numerose sperimentazioni svolte, soprattutto in Gran Bretagna.
Nel 1983 la Commissione della Comunita' Europea ha dato avvio al progetto MICROPROLOG, finalizzato a promuovere sperimentazioni, coordinate fra di loro, sull'uso di tale linguaggio nella scuola secondaria superiore di diversi paesi.
In questo lavoro si illustra la proposta di sperimentazione didattica sviluppata nell'ambito di tale progetto dal gruppo di Genova. Tale sperimentazione riguarda l'indagine di un problema reale: scelta di un prodotto in base ad esigenze prefissate, al fine di far acquisire agli studenti capacita' di tipo logico-deduttivo nella risoluzione di problemi.

INTRODUZIONE

Insegnare ad organizzare il pensiero e strutturare logicamente la conoscenza e' sempre stato un problema educativo di importanza rilevante; ma, poiche' i contenuti, le tecniche e i metodi di un sistema educativo cambiano o dovrebbero cambiare in relazione alle esigenze della societa', tale insegnamento e' avvenuto nel tempo per vie diverse. Per esempio, in passato, la capacita' di ragionare logicamente veniva indotta nei ragazzi attraveso lo studio delle lingue morte (greco e latino) o di alcune parti della matematica. Gli studi di tipo classico interessano oggi una netta minoranza di studenti, mentre capacita' di tipo logico-deduttivo devono far parte del patrimonio culturale di tutti; e' quindi necessario cercare delle forme alternative. L'interesse e l'entusiasmo dei ragazzi di fronte alle nuove tecnologie in generale e, in particolare, ai calcolatori sembra essere, attraverso la programmazione logica, un terreno promettente in questa direzione. I contributi piu' interessanti a questo riguardo sono descritti in [1] e in [3].

In base alla esperienza maturata nel settore delle Applicazioni del Calcolatore nella Didattica riteniamo che un linguaggio che deriva dagli studi sulla programmazione logica, quale il Prolog, oltre a contribuire a far acquisire una mentalita' di tipo causa-effetto nell'affrontare i problemi, offra altri vantaggi sia che si facciano scrivere

programmi ai ragazzi sia che li si facciano utilizzare. In primo luogo, la programmazione logica spezza quel rapporto privilegiato tra matematica e informatica che e' stato consolidato dall'uso dei tradizionali linguaggi di tipo algoritmico e dalle applicazioni di tipo numerico. Questo fatto e' positivo perche' pone tutta la classe docente sullo stesso piano di fronte al problema dell'introduzione delle nuove tecnologie; favorisce un rapporto di collaborazione tra gli insegnanti delle diverse materie, stimolandoli a trattare questo tema in modo interdisciplinare. Infine, l'uso di un linguaggio quale il Prolog consente di focalizzare l'attenzione su uno degli aspetti cruciali dell'insegnamento perche', indipendentemente dallo specifico contesto di ogni singola disciplina, permette di trattare in modo chiaro ed esplicito il problema della organizzazione e strutturazione della conoscenza. Anche gli allievi traggono vantaggio se non vedono l'informatica solo connessa alla matematica; infatti qualora si privilegino le applicazioni numeriche si corre il rischio che gli studenti riversino su questa nuova disciplina i pregiudizi che molti di essi hanno nei confronti della matematica. Si evita inoltre che gli strumenti di calcolo vengano associati soprattutto ad applicazioni di tipo numerico, quando oggi, invece, si diffondono quelle di tipo simbolico. Infine, si impedisce che gli studenti applichino tecniche e metodologie di carattere generale, acquisite attraverso la programmazione, solo all'analisi di problemi di natura matematica.

L'uso di programmi Prolog gia' predisposti consente di abituare gli allievi a riflettere su come ricavare informazioni da un insieme di dati precedentemente organizzati. Infatti un programma Prolog e' sostanzialmente una base di dati che occorre interrogare e non solo un programma da far eseguire dopo aver inserito i dati. In questo modo, percio', e' possibile educare a collegare informazioni specifiche ad altre piu' generali al fine di indagare una situazione, a stabilire operativamente nessi fra fatti di natura differente, a distinguere informazioni pertinenti da quelle irrilevanti, capacita' che oggi assumono un importante rilievo grazie alla diffusione delle basi di dati.

IL PROGETTO MICROPROLOG

In base a queste considerazioni abbiamo aderito con molto interesse alla proposta di inserimento nel progetto MICROPROLOG supportato dalla Comunita' Economica Europea, nel cui ambito ci si propone di realizzare esperienze sull'uso del linguaggio Micro-Prolog nella scuola secondaria superiore. Finora hanno aderito al progetto gruppi dei seguenti paesi: Belgio, Francia, Grecia, Italia.
Scopo del progetto e' l'analisi ed il confronto delle sperimentazioni didattiche realizzate da ciascun gruppo, al fine di studiarne l'efficacia come strumento per indurre

negli studenti capacita' di sviluppare ragionamenti di tipo logico-deduttivo.
Le diverse esperienze, che sono ora in atto, riguardano sia l'introduzione del Prolog come linguaggio di programmazione, sia la sua applicazione nell'insegnamento di altre discipline.

Nell'ambito del progetto MICROPROLOG il nostro gruppo si propone di condurre una sperimentazione in piu' classi di scuola secondaria superiore genovesi finalizzata ad utilizzare il linguaggio Micro-Prolog per insegnare ai ragazzi a risolvere problemi, insistendo su una impostazione di tipo logico della soluzione e sulla organizzazione della conoscenza. Il problema proposto agli studenti consiste nel determinare la configurazione di un personal computer che meglio risponde alle esigenze fissate.

Il problema proposto presenta alcune caratteristiche che lo rendono didatticamente significativo da diversi punti di vista. In primo luogo, esso suscita l'interesse dei ragazzi, perche' e' un problema attuale ed una esigenza reale che molti di essi hanno. Inoltre, permette di rinnovare i contenuti dell'insegnamento, perche' offre lo spunto per introdurre nei programmi alcune nozioni di informatica. Infine, la scelta di un prodotto in base ad esigenze prefissate richiede di organizzare informazioni e di sviluppare ragionamenti seguendo criteri indipendenti dallo specifico prodotto in esame: precisa formulazione delle esigenze, individuazione di fonti di informazione, analisi dell'informazione ivi contenuta, scelta di quelle pertinenti al problema, loro organizzazione in modo uniforme, sviluppo di un ragionamento in larga parte del tipo causa-effetto che permetta di risolvere il problema in modo soddisfacente.

In base a queste considerazioni riteniamo che il problema proposto faciliti il raggiungimento dei seguenti obiettivi didattici: abituare a pianificare azioni e stabilire quali sono i parametri che intervengono nella definizione del piano; abituare a documentarsi; educare alla lettura di informazioni di diverso tipo (testo scritto, grafici, etc.); indurre un atteggiamento critico e attivo nei confronti di fonti di informazione.

La proposta di lavoro e' articolata in cinque fasi successive illustrate in figura 1.

L'organizzazione della conoscenza, che costituisce l'aspetto centrale del lavoro, e' articolata in tre momenti successivi (v.fig.1). Poiche' l'esempio proposto ha scopo esclusivamente didattico, l'indagine e' limitata ad un basso numero di personal computer oggi disponibili sul mercato. Si ritiene pero' didatticamente importante che le fonti di informazione che gli studenti analizzano siano proprio i depliant che reclamizzano i prodotti, in quanto il processo

di lettura di informazioni, codificate secondo certi criteri, e la loro successiva riorganizzazione e codifica in base ai propri obiettivi, ha un notevole valore formativo. Si e' percio' scartata la scelta di fornire agli studenti informazioni gia' organizzate in tabelle quali sono quelle riportate sulle riviste di personal computer.
Dopo aver individuato alcuni parametri caratteristici quali, ad esempio, il costo, la espandibilita', la compatibilita' con altre apparecchiature, il software disponibile, la velocita', etc., gli studenti determinano quelli che influiscono sulla scelta in base alle loro esigenze. Successivamente, costruiscono le tabelle ad essi relative al fine di ottenere una prima rappresentazione della conoscenza.

La codifica delle informazioni in forma tabellare comporta come naturale conseguenza la descrizione in fatti Prolog utilizzando relazioni [2]. Attraverso questa costruzione gli studenti, in modo per essi assai naturale, giungono ad una strutturazione logica delle informazioni che hanno, ed a percepire concretamente i fatti che si possono dedurre dalle informazioni a disposizione a partire dalle proprie esigenze. Le regole vengono introdotte solo dopo che lo studente ha interrogato in vari modi la base di dati costituita solo dai fatti [3].

In ogni sperimentazione didattica la verifica delle capacita' acquisite e' un punto cruciale. Le verifiche iniziale e finale hanno lo scopo di fornire indicazioni utili alla valutazione dell' esperienza svolta. In particolare, la verifica iniziale e' finalizzata a verificare le capacita' di lettura, interpretazione, uso di informazione gia' codificata in modo omogeneo e ad uniformare la classe per quel che riguarda questi prerequisiti. La verifica finale ha lo scopo di valutare quanto il lavoro svolto abbia aumentato queste capacita' (v. fig. 1).

La scelta dell'esempio, o degli esempi, su cui far lavorare i ragazzi e' assai delicata. Problemi molto semplici, che richiedono di organizzare una conoscenza assai limitata, possono essere poco stimolanti: l'organizzazione puo' essere ovvia, e quindi non richiedere alcuno sforzo ai ragazzi. Problemi anche complessi, per i quali e' gia' nota una organizzazione della conoscenza, non rispondono agli obiettivi dell'esperienza, che si propone di far acquisire ai ragazzi capacita' di strutturazione della conoscenza. Un problema reale, espresso in modo informale, sembra essere la scelta piu' adeguata. La possibilita' di affrontarlo a livelli di complessita' differente consente di tener conto delle capacita' dei ragazzi e di procedere in modo graduale. Nel caso proposto, il numero ed il tipo di parametri che intervengono nella formulazione delle esigenze determinano la complessita' del problema.

CONCLUSIONI

La programmazione, come risulta dalle molte esperienze svolte, si e' rivelata un ottimo strumento come ausilio nel processo insegnamento/apprendimento, in particolare per quel che riguarda l'acquisizione di concetti base, quali ad esempio metodologie di risoluzione di problemi, visione computazionale di processi, e cosi' via.
La programmazione logica puo' favorire il recupero di abilita' di tipo logico-deduttivo, che, tradizionalmente introdotte per una via inadeguata alla odierna realta' scolastica, rischiano di diventare patrimonio di pochi, in netto divario con le esigenze di oggi.

RIFERIMENTI BIBLIOGRAFICI

[1] Ennals R.,Beginning micro-PROLOG, Ellis Horwood limited, England, 1983

[2] Kowalski R., Logic for Data Description, in Logic and Data Bases, Gallaire H. and Minker J. (Eds.), Plenum Press, USA, 1978, pp.77-103

[3] Kowalski R., Logic as a computer language for children, in New Horizons in educational computing, Yazdani M (Ed.), Ellis Horwood limited, England, 1984, pp.121-144

PROPOSTA DI LAVORO

FASE 1-VERIFICA DELLE CAPACITA' DI INGRESSO

- Test sulla capacita' di lettura di informazioni gia' organizzate;
- test sulla capacita' di organizzazione di informazioni.

FASE 2-ORGANIZZAZIONE DELLA CONOSCENZA

- Analisi di informazioni non codificate: analisi dei depliant illustrativi di diversi personal computers;
- introduzione di elementi di informatica;
- organizzazione della conoscenza prendendo in esame due parametri: costo ed espandibilta'.

FASE 3-TRADUZIONE IN UN PROGRAMMA MICRO-PROLOG

- Traduzione in fatti Micro-Prolog della conoscenza precedentemente organizzata;
- interrogazione della base di dati secondo una sequenza di domande di crescente difficolta': domande che non richiedono l'uso di variabili, domande che prevedono l'uso prima di una variabile e poi di piu' variabili, congiunzione di domande;
- scrittura di regole per rispondere a determinate domande.

FASE 4-RISOLUZIONE DEL PROBLEMA PROPOSTO IN SITUAZIONI PIU' COMPLESSE

- Organizzazione della conoscenza e scrittura di fatti e regole in Micro-Prolog prendendo in esame altri parametri, quali, ad esempio, la compatibilita' con altre macchine, il software disponibile e cosi' via.

FASE 5-VERIFICA DELLE CAPACITA' ACQUISITE

- Test sulle capacita' di lettura di informazioni;
- test sulle capacita' di organizzazione di informazioni.

Fig.1

DALL'ITALIANO AL PROLOG, USANDO IL PROLOG

D. Reboa, I. Prodanof*, G. Ferrari+

* Istituto di Linguistica Computazionale, Pisa
\+ Dipartimento di Linguistica - Universita' di Pisa

ABSTRACT

Viene presentata un'applicazione del Prolog al trattamento del linguaggio Naturale. Nella prima parte si introduce il formalismo Definite Clause Grammar (DCG), che traduce in modo efficiente ed aumentato una grammatica Context-Free (CF). Successivamente si presenta l'utilizzazione di variabili per la costruzione di strutture arbitrarie, a partire da una frase di input, esemplificata con un caso di traduzione di frase italiana in Prolog. Si prospetta infine l'utilizzazione di alcune estensioni delle DCGs.

1. Il Prolog e' giudicato per la sua qualita' di sottoinsieme della logica del primo ordine, un linguaggio promettente per l'implementazione di sistemi intelligenti, e quindi per molte applicazioni di Intelligenza Artificiale.

Nell'ambito del Natural Language Processing (NLP) non e' tuttavia ancora pienamente diffuso. A livello sperimentale il suo uso si e' dimostrato vantaggioso nella realizzazione di analizzatori di frasi (parsers). Questo perche'

i) fornisce la possibilita' di implementare in modo semplice e quasi immediato grammatiche, in particolare CF, per il trattamento di un sottoinsieme di fenomeni linguistici;
ii) permette di ridurre ad un solo passo, o meglio di compattare, l'analisi sintattica e l'interpretazione semantica, almeno per frasi di limitata complessita'.

Risale a Colmerauer (1975, 1978) la definizione di un formalismo logico (Metamorphosis Grammars - MG) per la descrizione del linguaggio naturale che trova in Prolog il miglior strumento implementativo; ad esso fece seguito la descrizione di un sottoinsieme minimale del francese (Colmerauer 1979), realizzata in Prolog tramite il formalismo delle MG da J.F. Pique (1978) ed utilizzata per creare e consultare data bases. Questo ispiro' i lavori successivi di V. Dahl (1977) per lo spagnolo, e di Pereira (1983) per l'inglese.

2.1. Nell'elaborazione di Pereira l'uso del Prolog permette di ridurre la realizzazione di un parser per il linguaggio naturale alla scrittura di una grammatica particolare, una Definite Clause Grammar (DCG).

Una produzione CF puo' essere considerata come l'espressione di una condizione di buona formazione relativa alla stringa di simboli presente nella parte destra della regola. Cosi' la regola CF

(1) S --> NP VP

puo' essere letta anche " la sequenza NP VP e' una stringa ben formata di tipo S ". Questo costituisce il passaggio immediato di una regola CF in termini di Definite Clauses. La produzione (1) puo' essere espressa come segue

(2) sentence(S0,S) :- noun_phrase(S0,S1), verb_phrase(S1,S).

che si legge "tra S0 e S si riconosce una frase se tra S0 ed S1 c'e' un sintagma nominale e tra S1 ed S c'e' un sintagma verbale". L'espressione (2) e' una non-unit clause ottenuta sostituendo i simboli atomici delle CFGs con simboli complessi costituiti dagli stessi simboli, con in piu' l'indicazione del punto di inizio e quello di fine di un costituente della frase. Cosi' i simboli della DCG diventano predicati Prolog a due argomenti. La regola CF del tipo

(3) determiner --> [the].

dove le parentesi quadre indicano che si tratta di un simbolo terminale, puo' essere espressa come segue

(4) determiner(S0, S) :- connects (S0, the, S).

che si legge "tra S0 ed S si riconosce un determiner se S0 e' collegato ad S mediante la parola the". Il predicato 'connects' e' usato per collegare le variabili che rappresentano i punti nella frase (delimitatori delle parole nella frase), con le parole che uniscono i punti.

Una grammatica CF espressa in termini di Definite Clauses viene eseguita come un programma Prolog e si comporta come un parser top-down per il linguaggio che la grammatica descrive. Infatti l'assiomatizzazione in Definite Clauses delle grammatiche CF permette di identificare gli algoritmi di parsing CF con la procedura di prova per la classe di Definite Clauses derivate dalle produzioni CF. Le regole della grammatica sono usate top-down, una alla volta, i goals in una regola sono eseguiti da sinistra verso destra; se ci sono regole alternative si puo' eventualmente tornare su esse mediante backtrack.

2.2. Le DCGs inglobano delle "augmentations" rispetto alle CFGs "ordinarie", eguagliando cosi' la famiglia delle Augmented Phrase Structure Grammars (APGS), di notevole successo nell'ambito del NLP.
Le DCGs estendono la notazione delle CFGs nel seguente modo:

- i non terminali possono essere termini composti, oltre che semplici atomi;
- nel lato destro di una regola, oltre ai non terminali e alle liste di terminali, ci possono essere chiamate di procedure racchiuse tra parentesi graffe.

Queste sono usate per esprimere le condizioni extra che devono essere soddisfatte affinche' la regola sia valida. Quindi il formalismo delle DCGs permette di:

- estendere una CFG con l'espressione delle dipendenze contestuali; ad esempio si possono aggiungere variabili per il controllo di genere e numero:

```
es: (5)
   noun_phrase(Num,Gen) --> det(Num,Gen),noun(Num,Gen).
   det(sing, masch)     --> [il ].
   det(sing, fem)       --> [la ].
   noun(sing, masch)    --> [gatto].
   noun(sing, fem)      --> [gatta].
```

Le variabili Num e Gen permettono di giudicare corretti i sintagmi nominali composti da articoli e sostantivi concordi in genere e numero.
- costruire strutture ad albero nel corso del parsing, indipendentemente dalle chiamate ricorsive della grammatica;

```
es: (6)
   sentence(s(NP,VP))    --> noun_phrase(NP), verb_phrase(VP).
   noun_phrase(np(Name)) --> name(Name).
   verb_phrase(v(V))     --> verb(V).
   name(name(paolo))     --> [paolo].
   verb(v(canta))        --> [canta].
```

la prima regola dell'esempio (2) stabilisce che una sentence con struttura

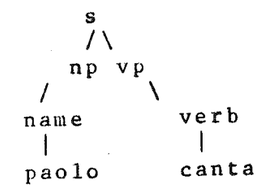
```
      S
     /  \
   NP    VP
```

e' fatta da un sintagma nominale con struttura NP, seguita da un sintagma verbale con struttura VP. Le clauses corrispondenti alle regole dell'esempio saranno

```
es: (6')
   sentence(s(NP,VP),S0,S)     :- noun_phrase(NP,S0,S1),
                                  verb_phrase(VP,S1,S).

   noun_phrase(np(Name),S0,S)  :- name(Name,S0,S).
   name(name(paolo),S0,S)      :- connects(S0,paolo,S).
   verb(verb(canta),S,S0)      :- connects(S0,canta,S).
```

A soddisfacimento avvenuto del goal sentence per la frase "paolo canta ", l'argomento di sentence sara' istanziato con la struttura



che e' l'albero di derivazione della frase.
- includere nelle regole della grammatica delle condizioni extra, che fanno dipendere il corso del parsing da computazioni ausiliarie; infatti si possono racchiudere tra graffe tests arbitrari.

es: (7)
```
data( Giorno, Mese ) --> mese(Mese),[Giorno],
                         {intero(Giorno), 0 < Giorno, Giorno < 32}.
```

La definite clause corrispondente all'esempio (7) e' la seguente

es: (7')
```
data(Giorno,Mese,S,S0) :- mese(Mese,S0,S1),
                          connects(S1,Giorno,S),
                          intero(Giorno),0 < Giorno,
                          Giorno < 32.
```

I predicati racchiusi tra le parentesi graffe sono lasciati inalterati.

In queste tre caratteristiche le DCGs sono compatibili con altri formalismi grammaticali tra cui soprattutto con le ATN di Woods, che sono un potente formalismo per l'analisi del linguaggio naturale, fondate anch'esse sulla "augmentation" di una CFG. Tuttavia le DCGs mantengono il carattere di dichiarativita' e quindi l'espressivita' e' maggiore (cfr. Pereira e Warren 1980).

3.1. Il nostro lavoro si propone come scopo di realizzare un parser per un sottoinsieme significativo dell'Italiano e la traduzione delle frasi italiane in predicati Prolog Si fonda sull'utilizzazione del formalismo delle DCGs che abbiamo gia' introdotto.

Come esempio illustrativo consideriamo il caso del riconoscimento di una frase dichiarativa in cui e' incassata una relativa esplicativa :

(8) "Paolo ascolta la radio che trasmette musica"

la traduzione proposta (Colmerauer 1979) e' la congiunzione di due fatti Prolog

(ascolta(Paolo, radio), trasmette(radio, musica))

La frase (8) e' riconosciuta dalla seguente grammatica CF

(9a) F --> SN SV


```
(9b) SN    --> NPRO F_REL
(9c) SN    --> ART  N F_REL
(9d) SV    --> V SN
(9e) F_REL --> che SV
(9f) F_REL --> nil
(9g) NPRO  --> Paolo
(9h) N     --> radio
(9i) N     --> musica
(9l) V     --> ascolta
(9m) V     --> trasmette
(9n) ART   --> la
(9o) ART   --> nil
```

in cui F sta per frase, SN sta per sintagma nominale, SV per sintagma verbale e F_REL per frase relativa.
In Prolog questa grammatica si trasforma, secondo i criteri visti sopra, nella seguente Definite Clause Grammar

```
(10a) f(S0,S)     :- sn(S,S1),sv(S1,S).
(10b) sn(S0,S)    :- npro(S0,S1),f_rel(S1,S).
(10c) sn(S0,S)    :- art(S0,S1),n(S1,S2),f_rel(S2,S).
(10d) sv(S0,S)    :- v(S0,S1),sn(S1,S).
(10e) f_rel(S0,S) :- connects(S0,che,S1),sv(S1,S).
(10f) f_rel(S0,S) :- succeed
(10g) npro(S0,S)  :- connects(S0,paolo,S).
(10h) n(S0,S)     :- connects(S0,radio,S).
(10i) n(S0,S)     :- connects(S0,musica,S).
(10l) v(S0,S)     :- connects(S0,ascolta,S).
(10m) v(S0,S)     :- connects(S0,trasmette,S).
(10n) art(S0,S)   :- connects(S0,la,S).
(10o) art(S0,S)   :- succeed.
```

Di fatto molti Prolog inglobano gia' dispositivi che facilitano la scrittura di una DCG. In questa sperimentazione particolare si e' usato lo Mprolog che permette la notazione con il simbolo -->, aggiunge automaticamente le variabili che rappresentano le posizioni e introduce automaticamente il predicato "connects".

3.2. Abbiamo gia' accennato che le DCGs consentono l'aggiunta di tests ausiliari.
La regola (9g) si potrebbe scrivere anche nel modo seguente:

```
(9ga)  npro(N)--> [N],{nome_p(N)}.
(9gb)  nome_p(paolo).
```

Cosi' invece di scrivere la regola (9g) per ogni nome proprio, si scrive una regola generica come la (9ga) e per ogni nome proprio si specifica il predicato 'nome_p'. Il vantaggio e' piu' evidente quando ci sono piu' condizioni da verificare.

4. La possibilita' di utilizzare variabili ausiliarie permette, come si e' visto, di costruire le strutture corrispondenti ad ogni simbolo atomico. E' possibile cosi' non solo costruire l'albero di parsing, ma qualunque struttura arbitraria non necessariamente aderente all'albero di parsing. Una caratteristica dei programmi scritti in Prolog e' quella di poter

restituire risultati incompleti; cioe' il termine o i termini che sono il risultato dell'esecuzione di una procedura (o del soddisfacimento di un goal), possono contenere variabili che sono istanziate successivamente con la chiamata di altre procedure. Tutte le occorrenze di una variabile sono istanziate simultaneamente. Cosi' la struttura della frase viene costruita pezzo per pezzo, lasciando come variabili le parti non ancora istanziate. Quando si rendono disponibili i frammenti necessari, i 'buchi', che nella struttura superiore sono ancora rappresentati da variabili, sono riempiti per unificazione.
Tra le strutture che si possono costruire c'e' anche la traduzione diretta della frase italiana in Prolog.

5. La grammatica che riconosce la frase (8) assume, quindi la seguente forma :

```
(11a)  f(R)                          --> sn(S,P,R), sv(S,R,P).
(11b)  sn(S,P,R)                     --> nome_p(S),f_rel(S,P,R)
(11c)  sn(S,P,R)                     --> art, nome(S), f_rel(S,P,R)
(11d)  sv((S,R,P)                    --> v(S,S1,P), sn(S1,P,R).
(11e)  f_rel(S1,P1,(P1,P2))          --> [che],sv(S1,R,P2)
(11f)  f_rel(S,P,P)                  --> [].
(11g)  nome_p(paolo)                 --> [paolo].
(11h)  nome(radio)                   --> [radio].
(11i)  nome(musica)                  --> [musica].
(11l)  v(S,S1,ascolta(S,S1))         --> [ascolta].
(11m)  v(S,S1,trasmette(S,S1)        --> [trasmette].
(11n)  art                           --> [la].
(11o)  art                           --> [ ].
```

Le regole di questa grammatica tradotte in Definite Clauses avra' in piu' i due argomenti che rappresentano i delimitatori della frase ed il predicato connects per i simboli terminali; secondo tale grammatica il goal da soddisfare per riconoscere e tradurre la frase (8) e' il seguente

f(R,[paolo,ascolta,la,radio,che,trasmette,musica],[])

il primo argomento conterra' alla fine la traduzione cercata.
Ignoriamo gli ultimi due argomenti che rappresentano gli estremi della frase e vediamo i passi dell'esecuzione del programma per il soddisfacimento del goal, esaminando l'espansione delle regole corrispondente.
Il goal iniziale f(R) porta all'espansione della regola (11a) che crea i due goals seguenti

sn(S,P,R),sv(S,R,P)

mediante la regola (11b) applicata al primo dei due goals, S viene istanziato a "paolo" (P ed R restano non istanziati); segue l'espansione del secondo con la (11d) che genera i due goals seguenti

v(paolo,S1,P), sn(S1,P,R)

Il soddisfacimento del primo goal mediante l'espansione con la (11l) determina l'istanziazione di P ad "ascolta(paolo,radio)".

Viene allora espanso il secondo goal con P istanziato, cioe' viene applicata al seguente goal

sn(S1,ascolta(paolo,S1),R)

la (11c) che produce i goals seguenti

art, nome(S1), f_rel((S1,R, ascolta (paolo,S1))

Il primo goal ha successo con l'applicazione della (11n), il secondo con la (11h) determinando l'istanziazione di S1 a " radio ".
L'ultimo dei tre goals si espande con la (11e) nel seguente modo

```
f_rel(radio, ascolta(paolo,radio),
              (ascolta(paolo,radio),P2))
    --> [che], sv(radio,R,P2).
```

La soluzione del goal sv(radio,R,P2) si ottiene sviluppandolo con la (11d); si generano cosi' i due goals seguenti

v(radio,S1,P2), sn(S1,P2,R)

Il soddisfacimento del primo goal con l'applicazione della (11m); cio' determina l'istanziazione di P2 a " trasmette(radio,S1) "
Il soddisfacimento del secondo goal con l'applicazione della (11c), della (11o) e della (11f) completa l'istanziazione di P2, con S1 istanziato a "musica".
Cosi' il terzo argomento di f_rel conterra' la congiunzione di fatti desiderata, che verra' passata a ritroso fino ad arrivare ad istanziare la variabile R della prima regola.

6. L'analizzatore che stiamo sviluppando tratta, per il momento, un insieme di frasi dichiarative con sintagmi nominali modificati da proposizioni relative .Per tutte queste si da' una traduzione in predicati Prolog. Per la soluzione di molti fenomeni si sono utilizzati i suggerimenti dati da Colmerauer (1979), ad esempio nella scelta di tradurre le relative come congiunzioni di fatti Prolog.
Il carattere strettamente dichiarativo delle DCGs, le rende tuttavia inadeguate al trattamento di alcuni fenomeni linguisticamente piu' complessi, come certi tipi di relative, le frasi con incassamenti e le interrogative.
Si intende integrare nella nostra grammatica il formalismo delle Extraposition Grammars (XGs) (Pereira 1981, 1983), che sono un ampliamento delle DCGs e facilitano il trattamento di fenomeni linguistici piu' complessi come certi tipi di relative e le interrogative.
La caratteristica saliente delle XGs e' la possibilita' di far apparire nella parte sinistra di una produzione piu' di un simbolo. Ad esempio, l'analisi della frase

qual e' il fiume che attraversa Trento ?

richiede il legamento tra "qual" e "fiume" per individuare il sintagma nominale.
Questo e' reso con una regola del tipo

```
q_marker... np --> int_det.
```

nella quale si esprime il fatto che prima di riconoscere il pronome interrogativo bisogna collegarlo ad un sintagma nominale che apparira' successivamente all'espansione di altre regole.
La traduzione proposta e' la seguente congiunzione di goals

```
? fiume(X),attraversa (Trento,X)
```

Bibliografia.

Bolc L. (ed.), Natural Language Communication with Computers, Springer Verlag, 1978.

Colmerauer A., Un sous-ensemble interessant du Francais, R.A.I.R.O. Informatique theoretique, vol. 13, n. 4, 1979.

Colmerauer A., Metamorphosis Grammars, in (Bolc 1978), 1975.

Dahl V., Un Sisteme Deductif d'interrogation de banques de donnees en Espagnol, Groupe d'Intelligence Artificielle U.E.R de Luminy, Universite' d'Aix-Marseille II, 1978.

Pereira F., Warren D. H., Definite Clause Grammars for Language Analysis. A survey of the formalism and a comparison with ATN, Artificial Intelligence 13 (1980).

Pereira F., Extraposition Grammars, AJCL 7 (4):243-256, 1981.

Pereira F., Logic for Natural Language Analysis, Technical Note 275, SRI International, January 1983.

Pique J.F., Interrogation en francais d'une base de donnees relationelles, Diplome d'Etude Approfondie, Groupe d'Intelligence Artificielle, Universite' d'Aix-Marseille, julliet 1978.

UN ANALIZZATORE MORFOLOGICO DELLA LINGUA ITALIANA

Silvana Leuzzi (*) Marina Russo (**)

(*) *Facolta' di Scienze dell'Informazione, Univ. Salerno*
(**) *Centro Scientifico IBM, Roma*

Introduzione

Nell'ambito di un progetto volto alla comprensione di testi scritti in linguaggio naturale, e' stato realizzato in VM/Prolog (1) un analizzatore morfologico della lingua italiana.
Le forme della lingua italiana vengono generate da una grammatica context-free (2) i cui elementi terminali sono i prefissi, radici, suffissi, alterazioni, desinenze e enclitiche e il cui assioma e' la *parola*.
Il lessico e i terminali della grammatica sono memorizzati in nove tabelle di una base dati relazionale (3), gestita dal sistema SQL/DS (4).
Sequenze fisse di parole variabili (tempi composti dei verbi) vengono riconosciute e analizzate per mezzo di una grammatica context-free il cui assioma e' il *tempo composto*.

Scelta di una struttura dati

Per una corretta analisi morfologica bisogna accedere alle informazioni inerenti le varie componenti della parola cioe' prefissi, radici, suffissi, alterazioni, desinenze ed enclitiche: questa mole di dati va organizzata per potervi accedere in modo corretto e veloce (un normale vocabolario contiene in media 50.000 lemmi e col tempo si arricchisce di neologismi e perde arcaismi). Tra i problemi del trattamento del lessico non c'e' pertanto solo quello delle sue dimensioni ma anche quello dell'aggiornamento.
La soluzione piu' semplice sarebbe quella di esprimere le informazioni sotto forma di fatti Prolog.
In questo modo si avrebbe una serie di fatti del tipo: *radice(dormire,dorm,3coniugazione)*. Tale predicato specifica che il lemma *dormire* e' un verbo della terza coniugazione con radice *dorm*.
Esprimere le informazioni in questo modo permette al programma di accedere direttamente ai fatti, data l'omogeneita' di programma e struttura dati, ma presenta l'inconveniente di non potersi avvalere di meccanismi di ricerca adeguati al trattamento di quantita' rilevanti di dati.
Si e' pertanto ritenuto opportuno separare la struttura dati dal programma e, in considerazione della mole dei dati da trattare, di avvalersi di una base dati relazionale.
E' stato usato come sistema d'accesso l'SQL/DS, che presenta il vantaggio di essere direttamente accessibile dal VM/Prolog tramite il predicato *sql(query,result,rc)*.
Il predicato Prolog "sql" valuta *query* come un'espressione computabile e passa il risultato all'SQL/DS per l'esecuzione, in seguito unifica la variabile libera *result* con il risultato e *rc* (numero o variabile libera) con il codice di ritorno dell'SQL.
Quando *query* e' una SELECT esso viene eseguito come un comando SQL e:
- se nessuna tupla e' stata trovata (rc = 100), il predicato fallisce,
- se la tupla e' stata trovata (rc = 0), gli elementi richiesti dalla select vengono prelevati dalla tupla e messi in una lista Prolog che viene unificata con *result*.
- se il backtracking rivaluta il predicato SQL viene cercata un'altra tupla che verifichi le precedenti condizioni.

I dati

Per una trattazione completa della morfologia sono state create nove tabelle in relazione a tutti i possibili controlli che potrebbero essere richiesti durante l'analisi di una parola.
Innanzitutto e' necessaria la costruzione di una tabella che contenga il **lessico** (l'insieme delle parole riconoscibili dal sistema) che viene utilizzata dall'analizzatore morfologico per effettuare l'ultimo controllo: anche se una parola e' stata riconosciuta dalla grammatica, non e' detto che essa appartenga al lessico italiano. Ad esempio, la parola *illimitezza* viene fatta derivare da *limite* aggiungendo un prefisso *il* e un suffisso *ezz*: il controllo finale nella tabella del lessico non trova *illimitezza*, il che significa che la parola non esiste o meglio, non appartiene al lessico del sistema.
Ci sono pero' dei casi in cui conviene effettuare subito il controllo di appartenenza della parola al lessico per evitare che venga fatta inutilmente l'analisi morfologica: la parola *con* (e piu' in generale tutte le parole invariabili) non e' accompagnata da caratteristiche morfologiche e non ha bisogno di un'analisi morfologica, ma solo di una ricerca nel lessico per verificare la sua esistenza e ricavare la sua categoria grammaticale.
Per questo e' stata aggiunta un'altra tabella e sono stati costituiti due tipi di lessici: uno denominato CAT_INVAR, contiene tutti i lemmi invariabili, l'altro denominato CAT_VAR, contiene quelli variabili.
I lemmi risultano cosi' divisi in due gruppi:
- quelli che in ogni caso devono essere scomposti per ricavare le caratteristiche morfologiche
- quelli di cui si deve ricavare solo la categoria grammaticale

In questo modo ogni parola che non si trova in CAT_INVAR deve essere analizzata.
Altre informazioni di cui ha bisogno l'analizzatore sono quelle relative alle locuzioni. A tal riguardo e' stata definita una particolare tabella, LOCUZIONI, in cui ogni tupla ha tre attributi: il primo contiene la locuzione, il secondo la parola meno frequente che compare nella locuzione, il terzo la categoria grammaticale associata alla locuzione.
Le altre tabelle sono state create per memorizzare gli elementi terminali della grammatica su cui vengono effettuati i controlli di esistenza.
I prefissi sono memorizzati in una tabella PREFISSI. La relazione e' costituita semplicemente da tuple con un unico attributo che contiene il prefisso. E' sufficiente utilizzare un unico attributo, dato che il prefisso serve solo a modificare la radice e non ha nessun'altra funzione.
Per le enclitiche viene utilizzata la tabella ENCLITICHE, in cui ogni tupla ha due attributi: il primo contiene la particella enclitica, il secondo le caratteristiche morfologiche della particella. La chiave di ricerca di questa tabella e' *enclitica*.
Prima di parlare delle altre tabelle e' necessario introdurre il concetto di classe desinenziale: una **classe desinenziale** e' l'insieme delle desinenze che sono in comune ad un insieme di morfemi e sono le uniche che possono seguire questi morfemi. Ad esempio, la classe desinenziale *dv_1coniug* e' l'insieme delle desinenze che possono seguire la radice di un verbo regolare della prima coniugazione.
RADICI e' la tabella che viene utilizzata per il trattamento delle radici. Ogni tupla di questa tabella ha quattro attributi: il primo attributo contiene il lemma da cui deriva la radice, il secondo attributo contiene la radice e costituisce la chiave di ricerca nella tabella, il terzo contiene la classe desinenziale della radice e l'ultimo attributo puo' contenere tre valori: 1, 2, 3 ciascuno dei quali individua il tipo di analisi da effettuare.
Le desinenze sono memorizzate nella tabella DESINENZE. Essa e' formata da tuple con quattro attributi: il primo attributo rappresenta una classe desinenziale, il secondo contiene una delle desinenze appartenenti alla precedente classe desinenziale, il terzo contiene le caratteristiche morfologiche relative alla desinenza di quella classe desinenziale, il quarto, infine, serve a guidare i successivi livelli di analisi. La chiave di ricerca di questa tabella e' la chiave composta *classe desinenziale* + *desinenza*.
Per i suffissi e' stata creata la tabella SUFFISSI in cui: il primo attributo contiene il suffisso vero e proprio, il secondo contiene il suffisso privato della desinenza, il terzo contiene la classe desinenziale. La ricerca viene effettuta sul secondo attributo.
La tabella ALTERAZIONI ha due attributi: il primo contiene l'alterazione privata della desinenza e il secondo la classe desinenziale relativa a quella alterazione. La chiave di ricerca e' *alterazione*.

Le fasi dell'analisi

L'analizzatore realizzato effettua su di una qualsiasi frase due tipi di analisi: **morfologica** (che tratta ogni singola parola) e hp2.morfosintattica. che serve a riconoscere le sequenze di parole che la sintassi deve considerare come un'unica espressione.
L'analisi morfosintattica e' stata a sua volta divisa in due fasi: **morfosintassi1**, che precede l'analisi morfologica e riconosce le sequenze invariabili di parole (locuzioni avverbiali, preposizionali, congiuntive) e **morfosintassi2**, che segue l'analisi morfologica e riconosce le sequenze variabili (tempi composti dei verbi, aggettivi superlativi e comparativi, ecc).
L'input dell'analizzatore e' costituito da una stringa di caratteri che rappresenta il testo da analizzare. Si presuppone che la frase in ingresso sia corretta morfologicamente (anche se il programma e' in grado di riconoscere parole con desinenze sbagliate, ogni parola che non e' presente nel lessico viene scartata e l'analisi passa alla parola successiva).
Un preanalizzatore trasforma la stringa di ingresso in una lista Prolog di **parole** (considerando parola tutto cio' che e' compreso tra spazi e segni di interpunzione). Suo compito e' quello di eliminare dalla lista tutti quegli elementi che non necessitano di una vera e propria analisi morfologica. La sua azione puo' essere suddivisa in tre fasi distinte:

1. eliminazione dei segni di interpunzione
2. eliminazione delle locuzioni (morfosintassi1)
3. eliminazione delle parole gia' analizzate in precedenza

La morfologia opera sulla lista cosi' trasformata, associando ad ogni parola una lista di elementi che contiene le caratteristiche morfologiche, la categoria grammaticale della parola, il lemma da cui deriva ed eventuali alterazioni ed enclitiche che ne possono far parte. I risultati di questa prima analisi vengono memorizzati in un file di appoggio che costituisce l'input della morfosintassi2.
La morfosintassi2 riconosce le sequenze variabili di parole (tempi composti dei verbi, superlativi e comparativi degli aggettivi, ecc.). Attualmente e' stata realizzata la parte relativa ai verbi composti che vengono trasformati in un unico elemento a cui e' associato il nuovo tempo del verbo.

Il preanalizzatore

Eliminazione dei segni di interpunzione

Tutti i caratteri speciali vengono direttamente eliminati dalla lista di ingresso (fatta eccezione ovviamente per l'accento, che si e' provveduto a differenziare dall'apostrofo). Poiche' tuttavia molti di questi caratteri sono necessari all'analisi sintattica e semantica (si pensi al $ o al %) vengono recuperati in seguito confrontando l'output dell'analizzatore con la stringa originale.

Morfosintassi1

La morfosintassi1 precede l'analisi morfologica per evitare che parole appartenenti a locuzioni siano analizzate inutilmente (all'interno della locuzione sono invariabili). Il riconoscimento delle locuzioni avviene secondo il seguente schema:

- per ogni elemento della frase si controlla se esso compare quale secondo attributo in qualche tupla della tabella LOCUZIONI
- in caso affermativo si verifica se la locuzione (il primo attributo della tupla) e' una sottosequenza della frase, effettuando un confronto tra la locuzione e la stringa in ingresso

- se la locuzione esiste realmente nella frase, si eliminano dalla lista di ingresso tutte le parole che compongono la locuzione e l'intera locuzione viene memorizzata assieme alle sue caratteristiche grammaticali (il terzo attributo della tupla) nel file che costituira' l'input per la morfosintassi2
- se uno dei due controlli precedenti da' esito negativo la ricerca si sposta sull'elemento successivo della frase fino a giungere alla fine della lista.

Ad esempio, data la lista *Mario.ha.studiato.fino.a.che.e(.giunta.la.sera.NIL* il preanalizzatore controlla se *Mario* e' il secondo attributo di qualche tupla, non trovandolo prosegue controllando se lo e' *ha* e cosi' via fino a quando trova che *fino* e' il secondo attributo della tupla:

fino a che	fino	congiunzione

verifica quindi se l'intera locuzione compare nella frase e, dato che la risposta e' affermativa, nel file di input per la morfosintassi2 viene inserito il record *fino a che.(congiunzione.fino a che.NIL).NIL.NIL.NIL.NIL* e la lista da analizzare diviene: *Mario.ha.studiato.e(.giunta.la.sera.NIL.*
Nel secondo attributo della tabella LOCUZIONI e' stata posta la parola meno frequente della locuzione, al fine di limitare al massimo il numero dei controlli: se ad esempio per la locuzione *fino a che* ci fosse *a* nel secondo campo della tabella, ogni volta che in una frase compare la preposizione *a* verrebbe inutilmente attivato il procedimento.
Il confronto tra la stringa in input e la locuzione e' stato realizzato mediante il predicato *cerca(*,*,*,*)*:

```
cerca(*locuzione,*input,*prima,*dopo)<-
          path(*locuzione,*input,nil,*prima,*dopo) & /().
path(*x.*l,*x.*m,*prova,*prova,*y)<-                & /().
          prefix(*l,*m,*y)
path(*loc,*t.*c,*proval,*tot,*y)<-                  &
          append(*proval,*t.nil,*prova2)            &
          path(*loc,*c,*prova2,*tot,*y)             & /().
prefix(nil,*y,*y)<-/().
prefix(*x.*l,*x.*m,*y)<-
          prefix(*l,*m,*y)                          & /().
```

Il predicato *prefix(*1,*2,*3)* unifica la lista *3 con *2 privata del "prefisso" *1. Il predicato fallisce se *1 non e' prefisso di *2.
Il predicato *append(*1,*2,*3)* concatena le due liste *1 e *2 e unifica *3 con la lista risultante.
Il predicato *path(*1,*2,*3,*4,*5)* riceve in ingresso la locuzione *1 da confrontare con l'input *2, utilizza *3 come variabile di appoggio e in caso che il confronto abbia successo unifica *4 con la lista delle forme di input che precedono la locuzione e *5 con la lista delle forme di input che la seguono.

Eliminazione delle parole gia' analizzate

E' stata creata in SQL la tabella FORME che contiene i risultati delle precedenti analisi morfologiche, al fine di ridurre al massimo i tempi di esecuzione e di mantenere un **dizionario** che viene costantemente aggiornato da un algoritmo di paginazione (attualmente in fase di realizzazione).
La tabella ha sei attributi, contenenti rispettivamente la forma analizzata, le sue caratteristiche grammaticali, il lemma da cui la forma deriva, le sue caratteristiche morfologiche e eventuali alterazioni e enclitiche.
Il preanalizzatore controlla se la parola da analizzare e' contenuta nel primo campo di una (o piu') tuple della tabella.

- In caso affermativo la parola viene eliminata dalla lista e viene inserito nel file di input per la morfosintassi il record ricavato dalla tupla estratta.
  Ad esempio, se una delle parole da analizzare e' *richiama*, nel suddetto file vengono inseriti i due record:

richiama.(vtr.richiamare.NIL).3.sing.pres.ind.NIL.NIL.NIL
richiama.(vtr.richiamare.NIL).2.sing.imp.NIL.NIL.NIL

- In caso negativo, il preanalizzatore prosegue con la parola successiva fino al termine della lista.

La morfologia

La morfologia costituisce il modulo fondamentale dell'intero programma e suo compito e' analizzare ogni elemento della lista precedentemente trattata dal preanalizzatore dando in output un file (CMORF PROLOG) contenente tutte le parole con i risultati dell'analisi.
Un modulo di riordinamento ristruttura il file in modo da presentarlo in ingresso alla morfosintassi2 con le parole nell'ordine in cui comparivano nella frase originale. Ogni record di questo file e' una lista relativa ad una parola o locuzione ed e' formata da cinque elementi:

1. parola
2. (categoria grammaticale . lemma da cui deriva . NIL)
3. caratteristiche morfologiche
4. (alterazione1.alterazione2. ... .NIL)
5. (enclitica1.enclitica2. ... .NIL)

Il programma effettua l'analisi morfologica di ogni elemento della lista. Per prima cosa controlla se l'elemento e' un invariabile, perche' in questo caso e' inutile applicare le regole della grammatica: la parola non ha caratteristiche morfologiche e non deve essere scomposta. La categoria grammaticale viene ricavata facendo un query sul secondo attributo della tabella CAT_INVAR che ha come primo attributo quella parola invariabile: *<-sql('select cat_gramm from cat_invar where lemma=parola_da_ricercare',*).*
Se la richiesta a SQL ha successo si aggiunge un fatto Prolog tramite il predicato *<-addax(fatto(si)).* per indicare che la parola e' stata gia' analizzata e quindi non si deve fare un'ulteriore analisi morfologica. Il programma, infatti, proseguendo controlla se nella workspace Prolog esiste il predicato *fatto(si).* e in questo caso passa alla parola successiva nella lista cancellandolo: *<-delax(fatto(si)).* per poter aggiungerlo in seguito senza che la successiva ricerca sia disturbata dalle precedenti: se *fatto(si)* non venisse ogni volta cancellato, l'analisi della parola successiva verrebbe inibita.
Ogni parola puo' essere costituita da un nucleo fondamentale, la radice, che puo' essere preceduta da un certo numero di prefissi e puo' essere seguita da un certo numero di suffissi, alterazioni ed enclitiche. In ogni caso dovra' essere seguita da almeno una desinenza che in alcuni casi puo' essere nulla (nomi indeclinabili: *citta'*).
Le leggi per la derivazione delle parole sono state formalizzate utilizzando una grammatica context-free:

$$PAROLA \text{ --> } parola\ invariabile$$
$$PAROLA \text{ --> } prefisso^n : TEMA : RESTO$$
$$TEMA \text{ --> } radice : suffisso^n : alterazione^n$$
$$TEMA \text{ --> } radice : desinenza : suffisso^n : alterazione^n$$
$$RESTO \text{ --> } desinenza : enclitica^n$$

dove x^n significa che x puo' essere ripetuto da 0 a n volte (puo' anche essere facoltativo) e il simbolo ':' indica il concatenamento di stringhe.
Il simbolo iniziale e' *'PAROLA'*, da cui si possono ricavare le "parole" di questa grammatica. Non tutte queste pero' sono parole della lingua italiana.
Ad esempio con questa grammatica viene riconosciuta la parola *bismettere* che non e' corretta da un punto di vista lessicale: dovra' percio' esistere un filtro che selezioni le parole relmente esistenti nella lingua italiana.

Il funzionamento dell'intero programma consiste nel fare delle ipotesi sulla composizione della parola da analizzare che viene scomposta in parti elementari procedendo da sinistra verso destra. Ogni parola viene infatti considerata come un blocco formato da un certo numero di componenti ad ognuna delle quali sono associate alcune condizioni: se la condizione relativa ad una componente viene soddisfatta si passa all'analisi della componente successiva, altrimenti l'intero blocco che rappresenta la parola viene scartato e si propone una nuova ipotesi.
Le condizioni non sono altro che dei test che vengono eseguiti sui valori degli attributi portati durante il lavoro di analisi. Nel caso di parole con un'enclitica, ad esempio, si deve controllare se la categoria grammaticale della sottostringa che precede la particella e' un *verbo* e solo in questo caso verra' continuata l'analisi (le particelle enclitiche possono appoggiarsi solo ad un verbo).
Per la parola *daglielo*, ad esempio, in base alla quarta produzione, viene formulata tra le altre l'ipotesi: *radice-desinenza-suffisso-alterazione*... Il programma riesce a riconoscere le prime due componenti che corrispondono alla sottostringa *da*, ma nel momento in cui controlla se *glielo* o una sua sottostringa sia un suffisso, fallisce. A questo punto viene percio' creata la nuova ipotesi: *radice-desinenza-enclitica* che portera' all'esatto riconoscimento di tutta la parola.
Se il precedente ragionamento venisse utilizzato in ogni caso, non verrebbero riconosciute correttamente parole come *mattone*, alle quali puo' venire attribuita anche la struttura: *radice-alterazione-desinenza* che considera *mattone* un "grande" matto. Per evitare queste interpretazioni errate e' stata utilizzata una struttura dati che permette di guidare l'analisi.
Si e' preferito dividere l'analisi morfologica in due fasi: analisi della parola senza i prefissi e analisi della parola con i prefissi. Questa scelta e' dovuta al fatto che le parole con prefissi sono poco frequenti ed e' quindi inutile iniziare l'analisi di una parola supponendo che possa avere un prefisso: suddividendo l'analisi della parola in questo modo si riesce a limitare il numero di accessi a SQL.
Se la parola non fosse analizzata in questo modo potrebbero sorgere altri problemi. Ad esempio la parola *bisogno* sarebbe riconosciuta come un "doppio sogno", perche' puo' essere decomposta in *bi-sogn-o* e niente puo' indicare che questa interpretazione e' sbagliata. Percio', se alla parola sono state assegnate delle caratteristiche morfologiche (il che equivale ad aver analizzato la parola) facendo una ricerca senza i prefissi, viene aggiunto il predicato *fatto(si)* per indicare che non si deve continuare la ricerca con i prefissi.

Ogni volta che viene trovata una corretta interpretazione per una parola, si fa fallire il predicato che vi ha portato in modo da attivare il backtrack del Prolog e far cosi' generare automaticamente tutte le possibili ipotesi per quella parola. Ad esempio per la parola *finissimo* una volta che e' stata generata l'interpretazione: *superlativo maschile singolare di fine* si fa fallire il predicato in modo che cerchi un'altra interpretazione. Prima cerchera' un'altra radice che possa avere come alterazione e desinenza *issimo* e trovera' *superlativo maschile singolare di fino* e dopo cerchera' un'altra interpretazione per la sottostringa *issimo* trovando per ultimo */ plurale imperfetto congiuntivo di finire*.
Si suppone che ogni parola in esame abbia tutte le componenti, ossia tutti gli elementi della grammatica formale prima definita: l'analizzatore pertanto richiama dei predicati per verificare se tali componenti esistono realmente. Ad esempio, nel predicato

```
parola1(*w,*x,*y,*lp)<-
        pref(*x,*u,*pref)          &
        parola1(*w,*u,*y,*ltot1).
```

si suppone che la parola abbia almeno un prefisso. Sara' il predicato *pref(*,*,*)* a decidere se l'ipotesi fatta e' corretta e solo in questo caso l'analisi procede sul resto della parola:


```
pref(*x,*y,*pref)<-
        stlen(*t,*1)                    &
        *m := *1 - 1                    &   /* almeno una radice */
        do(*x,*m,*lista1)               &
        firstn(5,*lista1,*lista,*)      &   /* un prefisso max 5 lettere*/
        proc(*lista,*list,'select pref from tabpref where pref in (''') &
        prefa(*y,*lista,*list,*pref)  &
        (¬var(*y) | *y = *x & *pref = nil) .
```

Si suppone che un prefisso possa essere composto al massimo da cinque lettere: e' sufficiente percio' controllare se tutte le sottostringhe delle prime cinque lettere della parola sono prefissi.
Se la parola contiene meno di cinque lettere e' inutile fare il controllo su tutte le sottostringhe, perche' almeno una lettera dovra' essere una radice, (non possono esistere parole senza radice): il controllo pertanto viene effettuato su tutte le sottostringhe della parola meno l'ultima.
*stlen(*1,*2)* e' una funzione di libreria che unifica *2 con la lunghezza della stringa *1.
Il predicato *firstn(*n,*1,*2,*3)* unifica *2 con i primi *n elementi della lista *1 e mette il resto in *3.
Il predicato *proc(*,*,*)* esegue i query a SQL.
Il predicato *prefa(*,*,*,*)* serve a stabilire quello che rimane della parola ancora da analizzare, una volta tolto il prefisso, basandosi sulla lista creata precedentemente dal predicato *do(*,*,*)* a partire dalla parola in esame: *storto* --> *(s.torto.NIL).(st.orto.NIL).(sto.rto.NIL).(stor.to.NIL).(stort.o.NIL).NIL.*
Quindi se e' stato trovato il prefisso *s* resta da analizzare la stringa *torto.*
L'analizzatore continua nella ricerca di una componente fino a quando fallisce, perche' non la trova nella base dati. Ad esempio, per la parola *storto*, una volta trovato il prefisso *s*, l'analisi continua chiedendo a SQL se *t, to, tor* e *tort* possono essere prefissi: poiche' nessuno di questi lo e', il predicato *pref(*,*,*)* fallisce e l'analisi passa a controllare se possono essere radici.
Per la parola *ristrutturare*, sia *ri* che *s* sono prefissi; ma quando l'analizzatore cerchera' di trovare una radice tra le possibili sottostringhe di *trutturare* fallira' perche' non ne esistono. In questo caso allora viene attivato il backtracking e la ricerca passa a considerare le possibili sottostringhe di *strutturare* che possono essere radici, trovando cosi' la radice corretta *struttur*.

La ricerca puo' inoltre essere guidata da alcuni tag contenuti nelle tabelle delle radici e delle desinenze.
Il primo controllo effettuato e' quello sull'attributo *ricerca* della tabella RADICI per mezzo del predicato *resto1* che verifica se i valori trovati dal predicato *radice* coincidono con quelli supposti. L'analizzatore in particolare fara' le seguenti scelte:

1	continua normalmente la ricerca.
2	(caso in cui la radice puo' essere seguita solo da una desinenza) richiama esplicitamente il predicato *resto4* che serve a riconoscere le desinenze.
3	(caso in cui non si deve fare backtrack sulle radici perche' ci si deve fermare a quella piu' lunga) mette un cut prima di continuare l'analisi in modo di non poter successivamente tornare indietro.

Un ulteriore controllo viene effettuato quando si incontra una desinenza per decidere cosa puo' seguirla, in base al valore che viene fornito dal predicato *desin(*,*,*,*,*,*)* In particolare:

no	l'analisi continua normalmente.
part_pass	(o **part_pres**) viene richiamato il predicato *resto2* che inizia la ricerca di nuovo dai suffissi.
pa	si evita di analizzare *sullo* come *su-l-lo* inserendo un cut dopo il predicato che riconosce la desinenza, in modo da non poter scomporre ulteriormente la desinenza *llo* riconoscendola come enclitica.

Nel secondo caso non viene richiamata direttamente l'altra desinenza dei participi perche' una desinenza participiale puo' essere seguita a sua volta da suffissi e alterazioni.
Ad esempio, la parola *sfacciataggine* e' formata da un participio, *sfacciat*, seguito da un suffisso e una desinenza, *aggine*.

La morfosintassi2

Questa fase deve riconoscere tutte le sequenze fisse di parole variabili quali i tempi composti dei verbi, gli aggettivi comparativi, le date, ecc.
La morfosintassi2 e' preceduta da un preanalizzatore che, basandosi sul file creato dalla morfologia, crea una lista in cui ogni elemento e' la lista delle possibili interpretazioni per una stessa parola.
Ad esempio, per la frase *uno stato libero* viene creata la lista:

(*uno* . (*art_deter* . *uno* .*NIL*) . (*sing.masch.*) .*NIL.NIL.NIL*).
(*uno* . (*agg_deter* . 1 .*NIL*) . (*sing.masch.*) .*NIL.NIL.NIL*).
(*uno* . (*pronome* . 1 .*NIL*) . (*sing.masch.*) .*NIL.NIL.NIL*).

((*stato* . (*nome* . *stato* .*NIL*) . (*sing.masch.*) .*NIL.NIL.NIL*).
(*stato* . (*vintr* . *essere* .*NIL*) . (*part.pass.sing.masch.*).*NIL.NIL.NIL*)).

(*libero* . (*agg_qualif* . *libero* . *nil*) . (*sing.masch.*) *NIL.NIL.NIL*).
(*libero* . (*vtr* . *liberare* . *nil*) . (1.*sing.pres.ind.*) .*NIL.NIL.NIL*).
NIL

Questa fase della morfosintassi e' descritta da una grammatica context-free il cui elemento iniziale e' *TEMPO COMPOSTO*.
Essa e' in grado di riconoscere tempi composti passivi (es: *erano state amate*), tempi composti attivi di verbi transitivi (es: *avevano amato*) e di verbi intransitivi (es: *erano andate*).
Le regole di produzione sono:

TEMPOCOMPOSTO --> *v.avere pptransitivo*
TEMPOCOMPOSTO --> *v.essere* : *RESTO*
RESTO --> *ppessere* : *pptransitivo*
RESTO --> *pptransitivo*
RESTO --> *ppintransitivo*

L'analizzatore entra in funzione ogni volta che incontra un participio passato e continua l'analisi in base al tipo di verbo incontrato.
Se ha trovato il participio passato del verbo *essere* controlla se questo e' seguito da un altro participio: verbo composto passivo del tipo *sono stato promosso*.
Se non e' seguito da un participio passato allora si e' nel caso di un tempo composto del verbo *essere*, ad esempio *sono stato*.
Se e' un participio passato di un qualsiasi altro verbo ed e' preceduto da una forma del verbo *avere* e' un tempo composto passato di un verbo transitivo, ad esempio *ho mangiato*, se e' preceduto da una forma del verbo *essere* e' un tempo composto passato di un verbo intransitivo, ad esempio *sono andato*, o un tempo composto passivo di un verbo transitivo, ad esempio *sono promosso*.


```
/********* verbi PASSIVI PASSATI (sono stato amato) *********************/
part(*t,*a.('vintr'.'essere'.nil).*d.*list,*f.*tot,*listagg,*tot,*1)<-
  substring(*d,*x,10,9)      &
  find(*f,*el,*x)            &
  ¬ *el = nil                &
  passivo1(*t,*a.('vintr'.'essere'.nil).*d.*list,*el,nil,*listagg,*t,*1).

/********* verbi INTRANSITIVI PASSATI (sono salito, sono stato) ********/
part(*t,*a.('vintr'.*x.nil).*d.*list,*tot,*listagg,*tot,*1)<-
  essere1(*t,*a.('vintr'.*x.nil).*d.*list,nil,*listagg,*t,*1) &
  (¬*listagg = nil | fail).

/********* verbi PASSIVI (sono amato) ***********************************/
part(*t,*a,*tot,*listagg,*tot,*1)<-
  passem1(*t,*a,nil,*listagg,*t,*1) &
  (¬*listagg = nil | fail).

/********* verbi ATTIVI PASSATI (ho amato) ******************************/
part(*t,*a,*tot,*listagg,*tot,*1)<-
  passatt1(*t,*a,nil,*listagg,*t,*1) &
  (¬*listagg = nil | fail).
```

Il predicato *part(*1,*2,*3,*4,*5,*6)* ha come input in *2 la lista delle caratteristiche morfologiche del participio passato da esaminare, in *1 la lista relativa alla parola che lo precede (verbo essere o avere), in *3 la lista relativa alle parole successive. Come output da' in *4 la lista delle caratteristiche morfologiche del tempo composto che ha trovato, e in *5 la lista delle parole successive ancora da esaminare; *6 e' una variabile di appoggio.
Il predicato *find(*,*,*)* controlla se l'elemento che segue il participio passato e' ancora un participio passato che concorda con il primo (confronta tra loro i campi che contengono le caratteristiche grammaticali e morfologiche).

Bibliografia.

(1) VM/Programming in Logic, **Program Description/Operation Manual,** SH20-6541-0, IBM Corp., 1985.
(2) D.E.Knuth, **Semantics of context-free Languages,** *Mathematical Systems Theory,* vol.2, 1968.
(3) C.J.Date, **An Introduction to Database Systems,** ed.Addison-Wesley, 1977.
(4) SQL/Data System, **Terminal User's Reference,** SH24-5017-2, IBM Corp., 1983.
(5) W.F.Clocksin, C.S.Mellish, **Programming in Prolog,** ed.Springer-Verlag, 1981.
(6) N.Chomsky, **Filosofia del linguaggio. Ricerche teoriche e storiche,** *Saggi linguistici,* ed.Boringhieri, 1969.
(7) N.Chomsky, **L'analisi formale del linguaggio,** *Saggi linguistici,* ed.Boringhieri, 1969.
(8) P.Tekavcic, **Grammatica storica dell italiano,V.2 Morfosintassi,** ed.Il Mulino, 1972.
(9) P.Tekavcic, **Grammatica storica dell italiano,V.3 Lessico,** ed.Il Mulino, 1972.
(10) M.Alinei, **La struttura del lessico,** ed.Il Mulino, 1974.
(11) S.Battaglia, V.Pernicone, **La grammatica italiana,** ed.Loescher, 1968.
(12) M.Dardano, P.Trifone, **La lingua italiana,** ed.Zanichelli, 1985.
(13) **La Grammatica. La Lessicologia,** *Atti del I e del II Convegno di Studi, Roma, maggio 27-28 1967,* ed.Mario Bulzoni, 1969.

UN LINGUAGGIO DI RAPPRESENTAZIONE DELLA CONOSCENZA BASATO SULLA PROGRAMMAZIONE LOGICA

Nicola Guarino (1), Fabio Santoro (2)

(1) LADSEB-CNR, Corso Stati Uniti 4 - 35100 Padova
(2) TECLOGIC s.c.r.l., Via Citolo da Perugia 68 - 35123 Padova

ABSTRACT

Viene qui discusso un formalismo per la rappresentazione della conoscenza, il Declarative Representation Language (DRL), che utilizza come linguaggio di implementazione il Prolog. Gli oggetti fondamentali trattati dal DRL sono concetti, istanze di concetti e relazioni. Tali oggetti vengono manipolati in modo omogeneo attraverso i due predicati Prolog "is_" e ":=", i cui argomenti vengono differenziati tramite opportuni operatori a seconda del significato.

Secondo l'approccio del Procedural Semantic Network (PSN) [1], cui il DRL si ispira, la conoscenza è strutturata sotto forma di rete semantica, nella quale i concetti possono essere organizzati in sotto-concetti ed essere a loro volta istanze di meta-concetti, dando origine ad una gerarchia ISA/INSTANCE-OF attraverso la quale è possibile ereditare parti, attributi e valori di attributi secondo regole distinte. Una tale organizzazione consente di trattare le procedure necessarie alla manipolazione della base di conoscenza in maniera omogenea a qualsiasi altro oggetto: un concetto viene manipolato attraverso i programmi ad esso associati, che possono essere ereditati, specificati in modo parziale o specializzati lungo gerarchie.

1.0 Introduzione

Il problema della rappresentazione della conoscenza costituisce sicuramente un aspetto fondamentale per la realizzazione di sistemi intelligenti: nei sistemi esperti della nuova generazione si sta diffondendo da una parte l'esigenza di adottare formalismi di rappresentazione della conoscenza sempre più potenti, dall'altra l'opportunità di considerare l'accesso ai dati di input come un "task" a sè, che può essere utilmente assegnato ad un sottosistema specializzato [2].

Presso l'Istituto per ricerche di Dinamica dei Sistemi e di Bioingegneria del CNR di Padova (LADSEB) è in corso un progetto per la realizzazione di un sistema esperto basato -tra l'altro- sulle considerazioni sopra esposte, finalizzato al trattamento delle aritmie

cardiache in unità di cura intensiva [3]. Un ambiente del genere è caratterizzato da una gran quantità di dati elementari, che richiedono interpretazioni semantiche anche complicate prima di poter essere utilizzati per il problema in oggetto.

Tra le tecniche di rappresentazione della conoscenza normalmente adottate in questi casi, quelle dei frame e delle reti semantiche sembrano essere le più promettenti; la loro utilità pratica (ed il loro interesse teorico) risentono però -oltre che della difficoltà di una implementazione efficiente- delle ambiguità epistemologiche [4] che si sono via via venute a creare, a partire dallo stimolante articolo di Minsky [5].

In questo lavoro ci si propone di esplorare le possibilità offerte dal Prolog (utilizzato come linguaggio di programmazione, quindi non solo come linguaggio logico) per l'implementazione di una rete semantica. Viene descritto a tale proposito il linguaggio DRL (Declarative Representation Language), che vorrebbe unire alla semplicità d'uso e di comprensione la coerenza e la potenza del formalismo di rappresentazione adottato.

Le idee fondamentali provengono sostanzialmente dal sistema OMEGA [6] e dall'approccio del Procedural Semantics Network (PSN) [1]; le modifiche e le estensioni derivano, oltre che dalla decisione di adottare il linguaggio Prolog, dalle precisazioni concettuali sviluppate da Brachman e colleghi a proposito del sistema KL-ONE [7] e dal formaslismo del "conceptual graph" introdotto da Sowa [8].

2.0 Individui, proprietà, classi, concetti.

Volendo utilizzare il linguaggio Prolog sia per la rappresentazione della conoscenza che per l'implementazione del motore di inferenza, è necessario, una volta definito in modo non ambiguo il significato dei termini utilizzati, distinguerli attraverso strutture simboliche opportune. Assumendo per il momento che ad ogni individuo della base di conoscenza sia possibile assegnare un nome unico, conveniamo innanzitutto di rappresentare con il termine Prolog "giovanni" l'individuo di nome Giovanni. Il predicato "is_", come vedremo, consentirà di definire, esplicitamente o implicitamente, gli individui conosciuti dal sistema, in modo da distinguerli dagli altri nomi simbolici utilizzati dal programma Prolog.

Fra tutti i predicati a un posto che potranno valere per tali individui, distinguiamo due categorie alle quali il progettista della base di conoscenza avrà arbitrariamente attribuito particolare valore semantico. La prima categoria riguarda quei predicati a un posto che possono essere interpretati come proprietà significative per alcuni individui, che chiameremo semplicemente proprietà. Ad esempio se vale:

```
ricco(giovanni).
```

potrà avere significato dire che "ricco" è una proprietà di giovanni, mentre se vale:

```
atom(giovanni).
```

il fatto che "giovanni" sia un atomo Prolog può non avere particolare valore semantico. Per distinguere i due casi e per consentire gli sviluppi successivi, che altrimenti neccessiterebbero di una logica del secondo ordine, scriveremo nel primo caso:

```
giovanni is_ ricco.
```

riservando l'usuale notazione solo ai predicati "comuni". Le espressioni così costruite [9] risultano tra l'altro molto vicine al linguaggio naturale; diremo che "giovanni" e "ricco" costituiscono rispettivamente il soggetto ed il predicato della dichiarazione "is_".

La seconda categoria è quella dei concetti. In generale ad ogni predicato ad un posto

(e quindi ad ogni proprietà), corrisponde una classe, eventualmente vuota,degli individui per cui tale predicato è soddisfatto, chiamata estensione del predicato (o della proprietà). Daremo il nome di concetti a quei predicati ad un posto che corrispondono a classi particolarmente significative, nel senso che tutti i loro elementi possono godere di una o più proprietà. Ad esempio se:

$$\forall \alpha \{ \text{uomo}(\alpha) \Rightarrow \text{mortale}(\alpha)\}.$$

può essere conveniente dire che "uomo" è un concetto, "mortale" una proprietà, e che il concetto "uomo" gode della proprietà "mortale". Utilizzeremo in questo caso il termine "a uomo" per rappresentare la generica istanza del concetto "uomo", e scriveremo:

a uomo is_ mortale.

Viceversa per esprimere il fatto che "giovanni" costituisce una istanza (specifica) dello stesso concetto scriveremo:

giovanni is_ a uomo. anziché giovanni is_ uomo.

Nel PSN per i concetti viene adottato il termine "classe", che noi invece riserveremo ad un qualunque insieme espresso in termini dei suoi elementi attraverso l'usuale notazione a lista: "[quaderno,penna,libro]" può rappresentare la classe corrispondente all'insieme degli oggetti che sono sul tavolo, che non si riferisce a nessun concetto particolare; "[giovanni,andrea,pierino]" può invece rappresentare (relativamente alla nostra base di conoscenza) l'estensione del concetto "uomo".

Prima di discutere le proprietà del predicato "is_" definiamo una terminologia comune per i termini fin qui introdotti (fig.1). Diremo che tutti i termini che entrano a far parte della base di conoscenza costituiscono dei riferimenti concettuali, nel senso che a questi, a differenza degli altri termini Prolog che possono comparire in un programma applicativo, viene attribuita un valore semantico particolare.

Nelle formule che compariranno nel seguito utilizzeremo le lettere greche α–θ per i vari tipi di riferimenti concettuali, che si dividono in due categorie: i riferimenti specifici compaiono come soggetto di una o più dichiarazioni "is_", mentre i riferimenti generici possono comparire come soggetto o come predicato. Dando ai vari termini della tassonomia di fig.1 il significato di meta-concetti, è possibile mantenere una meta-descrizione dei termini introdotti attraverso il predicato "is_", rimanendo all'interno di una logica del primo ordine; ad es.:

ricco is_ a property.
giovanni is_ a individual.
uomo is_ a concept.

Per coerenza occorre definire analogamente i concetti "property" e "individual" e lo stesso il concetto "concept"; se come nel PSN, introduciamo il concetto "object", le cui istanze sono costituite da tutti i concetti definiti, esplicitamente o implicitamente, nella base di conoscenza, possiamo scrivere:

property is_ a concept.
individual is_ a concept.
concept is_ a concept.
a concept is_ a object.

object is_ a object.

Per le dichiarazioni relative ad istanze generiche vale l'assioma:

(A1) $\forall \varepsilon, \gamma \{a\ \varepsilon\ \text{is\_}\ \gamma \Leftrightarrow \forall \alpha\ (\alpha\ \text{is\_}\ a\ \varepsilon \Rightarrow \alpha\ \text{is\_}\ \gamma)\}$

che deriva direttamente dalla definizione di relazione di inclusione tra insiemi. Da questo si

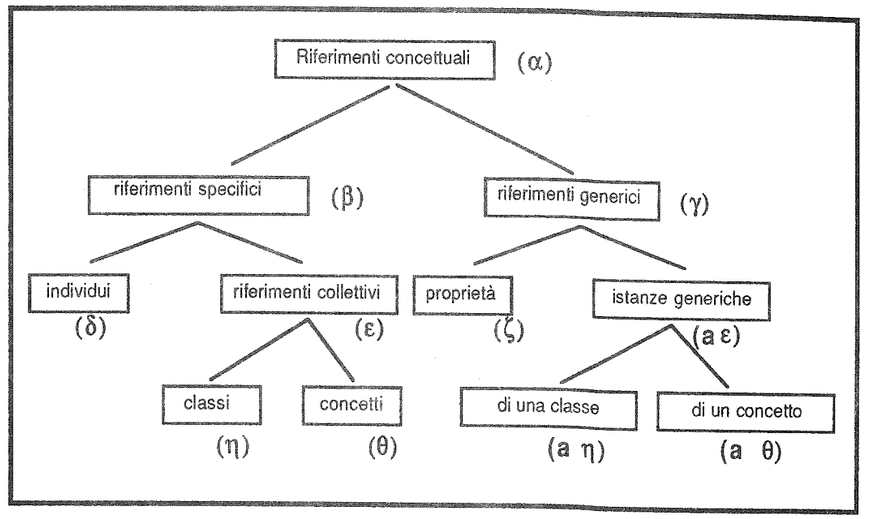


FIGURA 1

ricavano le proprietà:

1) $a\ \varepsilon\ \text{is\_}\ a\ \varepsilon$. 2) $(\alpha\ \text{is\_}\ a\ \varepsilon) \wedge (a\ \varepsilon\ \text{is\_}\ \gamma) \Rightarrow \alpha\ \text{is\_}\ \gamma$.

che chiameremo per comodità riflessiva e transitiva anche se non possono essere generalizzate per un qualunque riferimento concettuale. In particolare in base alla proprietà transitiva si può ricavare dalle definizioni precedenti:

giovanni is_ a student. ma non giovanni is_ a concept.

L'istanza generica di una classe può comparire come soggetto di una dichiarazione "is_" nel caso si voglia con un unico statement asserire l'esistenza di più istanze di un medesimo concetto:

a [pierino, gianni] is_ a uomo.

Mediante il predicato "is_" è possibil definire una tassonomia caratterizzata dalla presenza di gerarchie miste di tipo ISA ed INSTANCE-OF a più livelli, ai cui estremi sono definiti i concetti "object" e "nothing"; per quest'ultimo vale l'assioma:

$\forall\alpha\{a$ nothing is_ $\alpha\}$

3.0 Classi e concetti: il predicato ":="

Ad ogni concetto si può associare la sua estensione, cioè la classe costituita dalle sue istanze, attraverso il predicato di uguaglianza estensionale ":="; ad esempio supponendo che nella nostra base di conoscenza i tre colori possibili siano il rosso, il blu e il giallo, scriveremo:

(1) colore := [rosso, blu, giallo,verde].

Potranno essere definiti concetti di estensione nulla come:

nothing := [ ].

e altri la cui estensione è infinita o non nota; potranno viceversa esistere classi che non corrispondono all'estensione di nessun concetto, come "[pierino, gianni]" nell'esempio precedente. Nel seguito utilizzeremo impropriamente l'espressione 'la classe "colore" ' per riferirci all'estensione del concetto "colore".

Per renderci conto dell'opportunità dell'introduzione di questo predicato può essere interessante confrontare lo statement precedente con tre possibili dichiarazioni alternative di significato diverso, che utilizzano il predicato "is_ ":

(2) a colore is_ a [rosso, blu, giallo].
(3) a [rosso,blu,giallo] is_ a colore.
(4) [rosso, blu, giallo] is_ a colore.

La (2) e la (3) hanno un significato più debole della (1) e possono da queste essere dedotte in base all'assioma:

(A2) $\quad (\varepsilon_1 := \varepsilon_2) \Leftrightarrow (a\ \varepsilon_1\ is\_\ a\ \varepsilon_2) \wedge (a\ \varepsilon_2\ is\_\ a\ \varepsilon_1)$

che deriva dall'assioma di estensionalità della teoria degli insiemi; la (4) esprime, invece che una relazione di inclusione, una relazione di appartenenza della classe "[rosso, blu, giallo]" (considerata come individuo) a quella corrispondente al concetto "colore".

Il predicato ":=" costituisce una particolare relazione di uguaglianza, in quanto per esso valgono le usuali proprietà riflessiva, simmetrica e transitiva, mentre la proprietà di sostituzione vale solo per il predicato "is_" limitatamente ai suoi argomenti di tipo "a ε" (con il simbolo "~" si rappresenta la relazione di equivalenza logica):

(A3) $(\varepsilon_1 := \varepsilon_2) \Rightarrow (a\ \varepsilon_1\ is\_\ \gamma \sim a\ \varepsilon_2\ is\_\ \gamma) \wedge (\alpha\ is\_\ a\ \varepsilon_1 \sim \alpha\ is\_\ a\ \varepsilon_2)$

4.0 Gli operatori di unione, intersezione e complementazione.

Per le classi e i concetti gli operatori "and" e "or" consentono le usuali operazioni di uinione e intersezione di insiemi. Ad esempio il concetto "persona" può essere così definito:

persona := (uomo or donna).
persona := (animale and essere_pensante).

gli assiomi che valgono per tali operazioni si ricavano immediatamente dalla teoria degli insiemi:

(A4) $\quad \varepsilon_1 := (\varepsilon_2\ or\ \varepsilon_3) \Leftrightarrow (a\ \varepsilon_2\ is\_\ a\ \varepsilon_1) \wedge (a\ \varepsilon_3\ is\_\ a\ \varepsilon_1).$

(A5) $\quad \varepsilon_1 := (\varepsilon_2\ and\ \varepsilon_3) \Leftrightarrow (a\ \varepsilon_1\ is\_\ a\ \varepsilon_2) \wedge (a\ \varepsilon_1\ is\_\ a\ \varepsilon_3).$

in particolare la prima delle due dichiarazioni precedenti non implica che le estensioni dei due concetti "uomo" e "donna" siano disgiunte; ciò può essere espresso con la dichiarazione ulteriore:

(uomo and donna) := [].

L'operatore "not" consente l'operazione di complementazione, mediante la quale è possibile definire gli assiomi:

(A6) not nothing := object. (A7) not object := nothing.

5.0 Relazioni, attributi, parti.

Estendendo il discorso fatto per i predicati ad un posto a proposito di proprietà e concetti, diremo che tra tutti i predicati a due posti (o relazioni) che potranno valere per gli individui della nostra base di conoscenza, ne esistono alcuni -di particolare valore semantico- che possono essere interpretati in termini funzionali, come assegnazione di valore ad un attributo. Ad esempio questo può essere il caso della relazione

colore(mia_auto,blu). ma non della relazione 10 > 5.

Scriveremo nel primo caso:

colore of mia_auto := [blu].

utilizzando l'operatore "of" per definire un concetto derivato la cui estensione è costituita dai valori dell'attributo "colore" applicato all'individuo "mia_auto". Ciò consente tra l'altro di assegnare ad un attributo più valori:

colore of tua_auto := [rosso,giallo].

Se un attributo è definito per tutte le istanze di un concetto diremo che rappresenta un attributo strutturale , ed utilizzeremo un riferimento generico all'interno del costrutto "of"; nel nostro caso:

colore of a auto := [blu,rosso,giallo].
colore of a auto_ministeriale := [blu].

A livello di meta-descrizione, gli attributi sono istanze del concetto "attribute":

colore is_ a attribute of a auto.

I concetti (o le classi) che costituiscono il dominio ed il range dell' attributo "colore" possono essere espressi come come valori di attributi strutturali del concetto "attribute":

range of colore := [[blu,rosso,giallo]].
domain of colore := [oggetto_colorato].

Attributi e proprietà strutturali possono essere ereditati lungo le gerarchie ISA e INSTANCE-OF secondo le regole definite per il PSN, il quale distingue tra attributi e valori di attributi. I valori di default vengono trattati come particolari valori del meta-attributo "default"; ad esempio, la dichiarazione

default of lavoro of a figlio of a impiegato := [studente].

esprime il fatto che il valore di default dell'attributo "lavoro" per ogni istanza del concetto "figlio of a impiegato" è "studente". La classe "figlio of a impiegato" rappresenta l'immagine della classe "impiegato" secondo la relazione "figlio".

Una relazione particolarmente significativa è quella che lega un concetto alle sue parti costituenti. Utilizzeremo in questo caso l'attributo "part", e scriveremo, ad esempio:

part of a automobile := [carrozzeria,motore].
sedile is_ a part of a carrozzeria of a automobile.

Per il costrutto "is_ a part of" vale la proprietà transitiva, per cui si può dedurre:

sedile is_ a part of a automobile.

6.0 Programmi.

Come nel PSN, le modalità di manipolazione di ogni concetto possono essere definite in modo indipendente attraverso quattro programmi ad esso associati, che corrispondono alle quattro operazioni fondamentali fetch, test, add e delete. La prima consente di accedere alle istanze del concetto e la seconda di verificare le condizioni per cui un individuo può essere considerato una sua istanza, mentre la terza e la quarta sono relative alle usuali operazioni di inserimento e di cancellazione.

Ogni programma è un'istanza del meta-concetto "program", che possiede quattro attributi strutturali: "prerequisite", i cui valori rappresentano goal Prolog che devono essere soddisfatti prima che il corpo del programma venga eseguito; "body", il cui valore rappresenta il goal che costituisce il corpo del programma; "effect" e "complaint", il cui valore rappresenta un goal che viene eseguito nel caso che il "body" abbia successo o fallisca. Una tale organizzazione consente di specializzare i programmi lungo gerarchie allo stesso modo di qualunque altro oggetto; quattro programmi standard sono definiti come valori di default per il concetto "object".

7.0 I predicati "is_def" e "::=".

Una volta che è stata definita una tassonomia per mezzo del predicato "is_", può essere importante per l'utente inferire se tra due concetti esiste una relazione di "parentela" stretta, nel senso che uno è la più piccola generalizzazione dell'altro. Conviene allora introdurre il predicato "is_def", del tutto analogo a "is_" salvo che per la proprietà transitiva. Se come primo o secondo argomento di tale predicato compare un'espressione corrispondente a un'operazione di unione o intersezione (estensionale)

tra due concetti, è possibile distinguere queste ultime dalle corrispondenti operazioni intensionali. Ad esempio (cf. [8], pag. 83):

unicorn := [].
(cat or unicorn) := cat.
a (cat or unicorn) is_def a mammal.

Il predicato "::=" infine definisce la relazione di uguaglianza intensionale, per la quale non valgono le restrizioni dell'uguaglianza estensionale circa la proprietà di sostituzione. Ad esempio:

nonno ::= padre of a genitore.

8.0 Conclusioni.

Si sono descritte le caratteristiche fondamentali di un linguaggio le cui potenzialità sembrano degne di essere esplorate. In particolare l'introduzione del predicato di uguaglianza estensionale consente la presenza contemporanea delle assunzioni di "mondo chiuso" e "mondo aperto" nello stesso database ([10], pag.214), mentre l'interpretazione dei valori degli attributi come concetti e l'introduzione delle parti ampliano le già elevate potenzialità del PSN. L'introduzione del concetto di proprietà dovrebbe infine rendere possibile l'amalgamazione di linguaggio oggetto e metalinguaggio, secondo la filosofia del sistema OMEGA.

L'implementazione è tuttora in corso e non sembra presentare difficoltà sostanziali, grazie anche alla particolare versione di Prolog adottata, l' MPROLOG, soprattutto per quanto riguarda la versatilità dei predicati di built-in, la modularità e le possibilità di interfaccia con altri linguaggi o col sistema operativo.

BIBLIOGRAFIA:

[1] Levesque H., Mylopoulos J. : "A procedural semantics for semantic networks" , in N. V. Findler (ed.) , "Associative Networks: representation and use of knowledge by computers", Academic Press 79.
[2] Mittal S., Chandrasekaran B., Sticklen J. : "PATREC: a knowledge-directed database for a diagnostic expert system". IEEE Computer, 9-84.
[3] Guarino N., Bortolan G., Cavaggion C., Degani R. : "Towards an expert system for arrhythmia management" Computers in cardiology 85.
[4] Brachman R. J. : "What is-a is and isn't: an analysis of taxonomic links in semantic networks". IEEE Computer, 10-83.
[5] Minsky M.: "A framework for representing knowledge", in P. Winston (ed.), The psychology of computer vision, McGraw-Hill 75.
[6] Attardi G., Simi M. : "Metalanguage and reasoning across viewpoints", ECAI 84.
[7] Brachman R. J. , Schmolze J.G. : "An overview of the KL-ONE knowledge representation system" Cognitive Science, 2-85.
[8] Sowa J.F : "Conceptual structures: information processing in mind and in machine", Addison-Wesley 84.
[9] Sandewal E.: "Representing natural language information in predicate calculus", Machine Intelligence 6, 71.
[10] Kowalski, R.: "Logic for problem solving", North Holland 79.

ANALISI E PROPOSTE PER L'USO DEI LINGUAGGI LOGICI NEI SISTEMI DI CONSULTAZIONE

Cristina Bena, Giorgio Montini

Laboratorio di Intelligenza Artificiale
CSI-Piemonte - Corso U. Sovietica, 216 - TORINO

ABSTRACT

Si illustra l'utilizzo dei concetti di tassonomia ed oggetto e degli operatori sintattici strutturati come possibili soluzioni al problema di migliorare la dichiaratività dei programmi Prolog, in vista di applicazioni nella costruzione di basi di conoscenza in sistemi esperti. L'integrazione di un meccanismo di trattamento automatico della tassonomia nel dimostratore di teoremi permette al programmatore di disinteressarsi di alcuni problemi tecnici di efficienza del ragionamento e di avvantaggiarsi della tipizzazione delle variabili. L'introduzione di opportuni operatori consente di rendere la semantica delle regole indipendente dalla strategia di controllo e di incapsulare l'uso del cut in costrutti ad alto livello.

1. Introduzione

Da più parti è stato suggerito l'uso del Prolog come linguaggio di rappresentazione della conoscenza nei sistemi di consultazione. Ad una prima analisi della questione appare che un sistema Prolog, su cui siano stati sviluppati, ad esempio, strumenti di supporto alla consultazione più potenti degli usuali meccanismi di traccia (1), possa essere considerato un vero e proprio "shell" di sistema esperto (Clar82).
Una critica più approfondita evidenzia tuttavia che il raggiungimento degli obiettivi del linguaggio (dichiaratività e leggibilità di programmi) è ostacolato dalla mancanza di costrutti linguistici ad alto livello (il FORTRAN dei linguaggi logici ... (McDe80)). Per questo motivo chi utilizza il Prolog può essere indotto a definire regole in modo eccessivamente dipendente dalla strategia di controllo, ad introdurre espressioni oscure e poco dichiarative, allo scopo di risolvere problemi di rappresentazione o di efficienza, e ad applicare gli operatori extralogici, che pure risultano indispensabili, in modo non strutturato.
Nell'articolo si illustra l'utilizzo dei concetti di tassonomia ed oggetto e degli operatori sintattici strutturati come possibili soluzioni per migliorare la dichiaratività dei programmi.
I vantaggi principali di queste estensioni riguardano aspetti generali della programmazione logica, quali il rapporto fra la dichiaratività della conoscenza, l'efficienza dell'apparato deduttivo ed il controllo. Inoltre, le estensioni migliorano la qualità delle domande e delle spiegazioni del sistema, rendendole più aderenti all'organizzazione sistematica della conoscenza propria di molti domini.

---

(1) In questa discussione non verranno presi in considerazione problemi relativi all'uso di conoscenza incerta e di ragionamenti inesatti.

2. Tassonomia ed oggetti

Nella nostra proposta una base di conoscenza è descritta da un insieme di regole, che fanno riferimento a concetti (categorie semantiche) e ad oggetti. Le categorie semantiche sono organizzate in una struttura gerarchica, la tassonomia, che viene definita come una rete semantica semplificata composta di soli archi rappresentanti relazioni di sottoinsieme (archi "ss"). Gli oggetti vengono associati alle categorie semantiche attraverso operatori di "appartenenza" (eun, equivalente all'arco "isa" delle reti semantiche) e "non appartenenza" (non).
Esempi di dichiarazione:

```
tassonomia.                          oggetti.
  animale - gatto.                     piemonte eun regione.
  ente_pubblico - regione.             piemonte non regione_autonoma.
```

La relazione espressa dalla tassonomia definisce un reticolo completo il cui massimo è l'insieme universo ed il cui minimo è l'insieme vuoto. Nella dichiarazione della tassonomia sono perciò ammesse sottocategorie con intersezione non vuota, mentre sono vietati i cicli (propri). Ad esempio:

```
atto_pubblico - atto_amministrativo.
atto_unilaterale - atto_amministrativo.
```

è una tassonomia in cui l'intersezione delle sottocategorie atto_pubblico ed atto_unilaterale non è nulla.
Nelle regole si può assegnare un dominio di validità alle variabili associando ad esse una o più restrizioni tassonomiche. Ad esempio:

```
contrattovalidofra (C,X1,X2)  : -
        contraenti (C,X1,X2), daccordo (C,X1), daccordo (C,X2)
        :   C eun contratto, X1 eun soggetto, X2 eun soggetto.
```

2.1. Algoritmo di unificazione

L'algoritmo di unificazione è stato ampliato per considerare le restrizioni sulle variabili e le proprietà degli oggetti. A questo proposito introduciamo alcune definizioni. Nel corso della dimostrazione una variabile può essere istanziata ad un oggetto oppure non istanziata; fra le variabili non istanziate distinguiamo quelle vincolate da un insieme di restrizioni e quelle non vincolate.
L'estensione dell'algoritmo è necessaria per trattare le variabili vincolate, distinguendo i tre casi possibili.
Caso 1: una variabile vincolata ed una non vincolata.
Le due variabili vengono unificate e la risultante è equivalente alla variabile vincolata. Esempio:

```
A: ()      B: (eun(gatto))        unify (A,B) = (eun(gatto))
```

Caso 2: due variabili vincolate.
L'unificazione è possibile solo se gli insiemi definiti dalle restrizioni associate alle due variabili hanno intersezione non nulla. La risultante

sarà vincolata ad assumere valori nell'intersezione. Esempio:
A: (eun(animale_carnivoro)) B: (eun(felino))
unify (A,B) = (eun(felino))

Caso 3: una variabile vincolata ed una istanziata ad un oggetto.
Le variabili vengono unificate (e la risultante opportunamente istanziata) se l'oggetto soddisfa le restrizioni della variabile vincolata. Per dimostrare ciò l'interprete può far ricorso all'ereditarietà implicita nella struttura tassonomica. Esempio, se "silvestro eun gatto.":
A: (eun(felino)) B: silvestro unify (A,B) = silvestro

2.2 Deduzione di proprietà tassonomiche degli oggetti

Le categorie a cui un oggetto appartiene (o non appartiene) non sono necessariamente foglie della tassonomia. Inoltre la definizione di un oggetto è "dinamica": nel corso del ragionamento si può infatti dimostrare la sua appartenenza ad una categoria più specifica.
A questo proposito si è introdotto il predicato eun i cui due argomenti sono un oggetto ed una categoria semantica e le cui regole sono definite dal programmatore.
L'interprete, a seguito della dimostrazione di un teorema eun, aggiorna la lista delle categorie semantiche dell'oggetto: se, ad esempio, sappiamo che "silvestro eun felino" e dimostriamo il teorema "?- eun(silvestro, gatto)" possiamo aggiornare la descrizione dell'oggetto "silvestro" aggiungendovi la proprietà "silvestro eun gatto".
L'algoritmo di unificazione prevede l'invocazione automatica di un goal eun nel caso 3, quando le informazioni attuali non consentano di concludere che l'oggetto soddisfa le restrizioni richieste. Se, ad esempio, si ha che "A: (eun(gatto)) B: silvestro" e sappiamo che "silvestro eun felino", l'unificatore invoca la dimostrazione del goal "?- eun(silvestro, gatto)".

2.3 Visita efficiente di una tassonomia

L'integrazione di un meccanismo di trattamento automatico della tassonomia nel dimostratore di teoremi permette al programmatore di disinteressarsi di alcuni problemi tecnici di efficienza del ragionamento.
Un caso tipico è quello del percorrimento di un albero tassonomico in direzione top-down o bottom-up. Ad esempio, si abbiano una tassonomia strutturata come in fig. 1 e la regola "ogni animale carnivoro, se è più forte di un altro animale, desidera mangiarlo". Si abbia inoltre una serie di oggetti (fig. 2). Esaminiamo alcune soluzioni al problema di rappresentare queste informazioni e di effettuare ragionamenti su di esse.

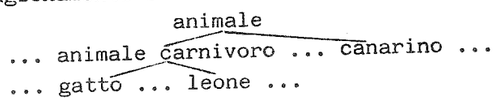


Fig. 1: tassonomia.

gatti = (..., silvestro, ...)
canarini = (..., titti, ...)

Fig. 2: oggetti.

Soluzione 1:

```
vuol_mangiare (A,B) :-
      animale_carnivoro (A), animale (B), molto_più_forte (A,B).
animale_carnivoro (A) :- gatto (A).
animale (B)           :- canarino (B).
gatto (silvestro).
canarino (titti).
```

La soluzione 1 (top-down), benchè descriva correttamente la semantica del problema, risulta inefficiente nel caso in cui si ricerchino nella rete le proprietà di un oggetto. Ad esempio, la query "?- vuol_mangiare (silvestro, titti)", percorre la tassonomia partendo dal nodo più generale (animale_carnivoro). Tale soluzione è inefficiente poichè il branching factor dello spazio degli stati risulta essere molto elevato: il numero di passi è linearmente dipendente dalla dimensione della tassonomia.
Qualora la ricerca fosse partita dalle proprietà dell'oggetto (gatto), il numero di passi sarebbe stato uguale alla distanza della proprietà dal nodo goal (2). Si può allora individuare la seguente soluzione 2 (bottom-up):

Soluzione 2:

```
vuol_mangiare (A,B) :-
      isa (A, animale_carnivoro), isa (B, animale),
      molto_più forte_(A,B).
is_a (silvestro, gatto).
is_a (titti, canarino).
s_s (gatto, animale carnivoro).
s_s (canarino, animale).
isa (A,B) :- is_a (A,B).
isa (A,B) :- is_a (A,C), ss (C,B).
ss (A,B) :- s_s (A,B).
ss (A,B) :- s_s (A,E), ss (E,B).
```

In questo caso, supposto di avere un'efficiente implementazione della ricerca nel database Prolog, il sistema trova sempre la soluzione in tempo minimo. Tuttavia si può trovare un ulteriore controesempio: "?- vuol_mangiare (A,B)". Il sistema trova le soluzioni scandendo ad uno ad uno tutti gli oggetti memorizzati nelle clausole is_a e verificando per ognuno l'appartenenza alle categorie semantiche animale_carnivoro ed animale. La complessità della soluzione è quindi lineare rispetto al numero di oggetti memorizzati nelle clausole is_a. Una strategia top-down avrebbe invece selezionato direttamente tutti gli animali carnivori estraendoli dal database.
Una possibile soluzione consiste nell'esame degli argomenti della query e nella scelta dinamica del miglior metodo di ricerca (top-down o bottom-up):

```
isa (A,B) :-     var (A),    !,    "strategia top-down".
isa (A,B) :-  nonvar (A),    !,    "strategia bottom-up".
```

---

(2) Nell'esempio si supponga che la tassonomia sia un albero proprio e che non vi siano pertanto intersezioni non vuote fra sottoalberi distinti.

2.4 Uso della tassonomia per fornire risposte generali

La nostra proposta, tuttavia, non risolve soltanto un problema di efficienza nel percorrimento di una tassonomia. Il meccanismo delle restrizioni sulle variabili permette infatti di avere risposte semanticamente più ricche, applicando idee sviluppate nelle ricerche sulla valutazione simbolica di programmi.
Il problema visto nel paragrafo precedente viene descritto nel nostro linguaggio nel seguente modo:

```
tassonomia.
  animale - animale_carnivoro.
  animale - canarino.
  animale_carnivoro - gatto.

oggetti.
  silvestro eun gatto.
  titti eun canarino.

regole.
  vuol_mangiare (A,B) :-  molto_più_forte (A,B)
      : A eun animale_carnivoro,  B eun animale.
```

La risposta alla query "?- vuol_mangiare (silvestro, titti)" viene risolta, come nella soluzione 3, applicando una strategia bottom-up. Il sistema risolve invece il goal "?- vuol_mangiare (A,B)" cercando di associare le variabili a restrizioni il più possibili generali e di evitare perciò di legarle a singoli oggetti: in questo caso una possibile risposta è "A eun gatto, B eun canarino".
I vantaggi di questo approccio sono notevoli:
- si può generalizzare la dimostrazione di un teorema ad intere categorie semantiche senza doverlo dimostrare per ciascuna istanza, ma ricorrendo ad una forma di valutazione simbolica;
- una dimostrazione generale è assolutamente indipendente dal numero di oggetti del database coinvolti;
- è possibile esprimere in modo effettivamente dichiarativo le relazioni tassonomiche tra i concetti.

Lo svantaggio principale, consistente nell'appesantimento dell'algoritmo di unificazione, può essere compensato sia da un'operazione di preprocessing della base di conoscenza, sia da un oculato utilizzo della tassonomia.
Si osservi, infine, che la risposta fornita dal nostro sistema è una generalizzazione delle risposte di Prolog standard; applicando un opportuno operatore di particolarizzazione è quindi possibile ricondursi al modello classico, senza per questo perdere i vantaggi computazionali dell'approccio più generale.

3. Operatori strutturati

Il programmatore Prolog è spesso portato a scrivere programmi la cui correttezza è fortemente dipendente dalla strategia di controllo: l'esempio classico viene dall'uso dell'operatore cut insieme con la strategia di controllo depth-first. Lloyd (Lloy84) sostiene che il cut in realtà non modifica la semantica dichiarativa dei programmi; esso piuttosto può introdurre dell'incompletezza nella procedura di dimostrazione, dovuta all'effetto di pruning dell'albero delle soluzioni.

D'altro canto, analizzando più a fondo il funzionamento del cut in una strategia depth-first, si può osservare come il suo utilizzo soddisfi alcune esigenze del programmatore della base di conoscenza:
- conferma della scelta di una regola: il cut separa le condizioni di selezione della clausola dagli antecedenti veri e propri; se la valutazione giunge all'operatore cut, la clausola considerata è quella corretta per la dimostrazione del goal; in questo caso, però, se il programma è dichiarativamente corretto, l'uso del cut è pericoloso in quanto può portare all'incompletezza della dimostrazione:
- definizione di casi particolari e di eccezioni a regole generale. Spesso un programmatore Prolog scrive una serie di clausole concernenti i casi eccezionali ponendole prima delle regole che descrivono il caso generali, e ricorre al cut per impedire che il processo di backtracking possa portare ad un'errata applicazione di queste ultime; questo è un tipico uso improprio del cut, in quanto programmi proceduralmente corretti non risultano tali dal punto di vista dichiarativo.

L'introduzione di predicati di ordine superiore, quali if-then-else, porta a programmi la cui semantica dichiarativa riflette in modo più accurato la relazione da calcolare, e che possono risultare ragionevolmente efficienti. Un problema aperto è in effetti quello di definire ed implementare correttamente tali costrutti. E' necessaria un'analisi non superficiale del problema: implementazioni dell'if-then-else quali la seguente:

```
ifthenelse (A,B,C) :- A, !, B.
ifthenelse (A,B,C) :- C.
```

non sono infatti corrette, come appare dal seguente esempio:

```
a (x).
b (y).
c (y).
?- ifthenelse (a(y), b(y), c(y)).
yes.
?- ifthenelse (a(X), b(X), c(X)).
no.
```

L'esempio mette in evidenza come avendo una query generale, cioè contenente variabili, esistono sia istanze che soddisfano la condizione A, sia istanze che non la soddisfano, ma che risultano a loro volta soddisfatte dalla condizione C e che in questa implementazione vanno perse.
L'approccio alternativo da noi seguito prevede l'introduzione di opportuni operatori sintattici, che consentono di esplicitare i vincoli di attivazione delle regole e di renderli indipendenti dalla strategia di controllo. Un predicato viene definito attraverso un insieme di microregole che il sistema compone durante una fase di compilazione in base alle direttive specificate dalle proprietà tassonomiche delle variabili e dagli operatori sintattici, in modo da formare un insieme di clausole Prolog.
Lo schema di dimostrazione di un goal prevede l'esame di tutte le microregole relative al predicato, l'individuazione di quelle applicabili e la loro valutazione, assumendo che l'and logico dei rispettivi antecedenti definisca una soluzione per il goal. Il non-determinismo può essere espresso sia implicitamente mediante la tassonomia, sia in modo esplicito, mediante un operatore.

Il confronto fra l'ambiente del goal ed il dominio delle variabili è il meccanismo di determinazione dell'applicabilità di una regola: una regola è applicabile se il suo dominio unifica con l'ambiente del teorema. Esempio:

```
1. carta_identita_valida (P) :-
        comune_residenza (P,C), carta_rilasciata (P,C)
        : P eun persona, C eun comune.
2. carta_identita_valida (P) :- firma_genitori (P) : P eun minorenne.
```

Se si chiede di dimostrare il teorema "?- carta_identita_valida (P) : P eun maggiorenne" il sistema determina come applicabile la sola regola 1, mentre per il teorema "?- carta_identita_valida (P) : P eun minorenne" vengono selezionate entrambe le regole 1 e 2. Il teorema "?- carta_identita_valida (P) : P eun persona" viene dimostrato spezzando l'ambiente in due sottoambienti, uno relativo alle persone minorenni, l'altro alle persone non minorenni. (3)
Infine, qualora nessuna delle microregole relative ad un predicato sia applicabile, il sistema assume che gli argomenti del teorema siano fuori del dominio di competenza del predicato: la dimostrazione pertanto fallisce. Nell'esempio precedente, la richiesta "?- carta_identita_valida (fido)" fallisce, posto che "fido eun cane".
Con l'operatore sintattico strutturato nelcasodi il programmatore può definire l'applicabilità di una regola, non solo mediante proprietà tassonomiche, ma anche con predicati. Sintatticamente le regole nelcasodi sono microregole il cui body è preceduto dalle precondizioni di selezione (questa sezione della regola è detta selettore), racchiuse dalle due parole chiave nelcasodi e allora. Esempio:

```
controllato (D) :-
        nelcasodi richiesti_chiarimenti (D,O)
        allora spedita_risposta (D)
        : D eun delibera, O eun organo_di_controllo.
```

Le regole nelcasodi vengono preprocessate in modo da partizionare l'ambiente in tre sottoambienti.
1. Un sottoambiente che non soddisfa le condizioni di selezione (dominio delle variabili e selettore); in questo caso la regola non è applicabile.
2. Un sottoambiente che soddisfa le condizioni di selezione ma non il body; per questo secondo sottoambiente la dimostrazione fallisce.
3. Un sottoambiente che soddisfa sia le condizioni di selezione sia il body.
In fase di esecuzione i sottoambienti vengono descritti dalle restrizioni sulle variabili.
Un'estenzione di questo costrutto permette di definire eccezioni ad una regola: le regole "nelcasodi ... deroga ..." e "nelcasodi ... in deroga a ... allora ..." permettono di definire i casi in cui una microregola generale non deve essere applicata ed eventualmente va sostituita con una microregola più particolare.

4. Conclusioni

Il problema di integrare le reti semantiche in un linguaggio logico del prim'ordine è stato affrontato da vari autori.

---

(3) Si osservi che nella regola 1 la variabile C, non essendo un argomento del conseguente, non interviene nel meccanismo di selezione.

In particolare in (McSk79) si introduce l'uso delle categorie semantiche e delle restrizioni sulle variabili, attraverso una modifica della Π-notazione estesa di (Fish75) ed un ampliamento dell'algoritmo di unificazione di Robinson.
Si trova un fondamento logico al nostro approccio anche nei lavori che trattano l'introduzione dei tipi in Prolog: oltre a (Mish84) e (Mycr84) occorre citare il lavoro di (Kana85) sulla inferenza dei tipi, in cui meccanismi di restrizioni tassonomiche sulle variabili vengono applicati alla dimostrazione della correttezza dei programmi.
L'esigenza di costrutti ad alto livello si riscontra in molto interpreti Prolog (ad esempio: bag-of, set-of, if-then). La necessità di coprire soprattutto il cut mediante predicati ad alto livello è stata sottolineata in (Lloy84).

Le proposte presentate in questo articolo sono in corso di sperimentazione nell'ambito del progetto ASPERA, per la costruzione di un sistema di ausilio all'esecuzione di procedure amministrative; il linguaggio di rappresentazione della conoscenza utilizzato è ispirato alle idee presentate in questo articolo e le integra in un modello contenente le microregole come struttura di base, una tassonomia, un meccanismo di restrizioni di variabili e dei costrutti di composizione, quali "nelcasodi" e "deroga".
Al progetto ASPERA collaborano il prof. F. Sirovich del Dip. di Informatica dell'Università di Torino, il dott. F. Massacesi e la dott.sa D. Formento della Regione Piemonte; contributo indispensabile nella determinazione dei contenuti e nella conduzione del progetto è prestata da Dario De Jaco, responsabile del Laboratorio di Intelligenza Artificiale del CSI-Piemonte; utili discussioni si sono avute con il dott. G. Rossi, il prof. G. Lolli ed il prof. A. Martelli del Dip. di Informatica dell'Università di Torino.

BIBLIOGRAFIA

(Clar82) - Clark, K.L., McCabe, F.G. "PROLOG: A Language For Implementing Expert Systems" in Machine Intelligence 10, 455-470, Ellis-Horwood 1982.
(Fish75) - Fishman, D.H., Minker, J. "Π-Representation: A Clause Representation For Parallel Search", Artificial Intelligence, 6 (1975), 103-127.
(Kana85) - Kanamori, T., Horiuchi, K., "Type inference in Prolog and its Application", Proc. IJCAI 1985, 704-707.
(Lloy84) - Lloyd, J.W. "Foundations Of Logic Programming", Technical Report 82/7 Revised March 1984, Dept. of Comp. Sci., University of Melbourne.
(McDe80) - McDermott, D. "The PROLOG Phenomenon", SIGART Newsletter, 72 (1980) 16-20.
(McSk79) - McSkimin, J.R., Minker, J. "A Predicate Calculus Based Semantic Network for Deductive Searching" in Associative Networks, ed. Findler, Academic Press (London) 1979.
(Mish84) - Mishra, P., "Towards a Theory of Types in Prolog", Proc. 1984 Int. Symp. on Logic Programming, 289-298.
(Mycr84) - Mycroft, A., O'Keefe, R.A. "A Polymorphic Type System for Prolog", Artificial Intelligence, 23 (1984), 295-307.

TECNICHE PER L'USO DEL PROLOG NELLA REALIZZAZIONE DI SISTEMI ESPERTI

L.Console, A.Martelli, G.Rossi

Dipartimento di Informatica, Universita' di Torino
Via Valperga Caluso, 37 - 10125 TORINO

SOMMARIO

In questo lavoro vengono descritti i risultati ottenuti in un progetto sull'uso del Prolog nello sviluppo di sistemi esperti. Risultato principale del progetto e' la definizione ed implementazione di uno schema di rappresentazione della conoscenza che consente di combinare, in modo flessibile e naturale, frame, regole di produzione e Prolog. Tutto il sistema e' implementato in Prolog, utilizzando una tecnica di preprocessing per generare l'effettiva implementazione Prolog a partire da un linguaggio di piu' alto livello con cui viene rappresentata la conoscenza sul dominio in esame. In questo lavoro verranno dapprima individuate le tecniche normalmente utilizzate nella realizzazione di sistemi esperti in Prolog, quindi verra' mostrata l'applicazione della tecnica di preprocessing proposta su un esempio relativo alla definizione di sistemi di regole di produzione con diverse strategie di inferenza e, successivamente, verra' descritto il funzionamento del sistema a frame realizzato. Verra' infine accennato a lavori in corso per estendere e migliorare lo schema proposto e per definire schemi diversi, basati su un modello ad oggetti.

1. INTRODUZIONE

Scopo di questo progetto e' l'analisi e la definizione di tecniche e strumenti per un uso effettivo del Prolog nella realizzazione di Sistemi Esperti.

L'impiego del Prolog in questo tipo di applicazioni e' stato oggetto di un crescente interesse in questi ultimi anni. Gia' esistono vari sistemi funzionanti implementati in Prolog, ma ancora si e' lontani da un comune accordo sulla sua adeguatezza per questo tipo di applicazioni.

Come primo passo abbiamo percio' analizzato le varie proposte di uso del Prolog per la realizzazione di sistemi esperti esistenti in letteratura ed abbiamo individuato tre diversi approcci [1]:

- uso diretto del Prolog, in cui vengono sfruttate direttamente le capacita' di rappresentazione e di inferenza del linguaggio stesso;
- uso indiretto, nel senso che il Prolog viene utilizzato come linguaggio di implementazione di un interprete per un sistema di elaborazione della conoscenza che puo' essere piu' o meno diverso dal Prolog stesso.
- estensioni e/o modifiche al Prolog in modo da renderlo piu' adeguato a questo tipo di applicazioni.

Nel primo caso il Prolog puo' essere convenientemente utilizzato come un

semplice sistema a regole di produzione, il cui meccanismo di inferenza e' l'interprete Prolog stesso. Il Prolog e' comunque sufficientemente potente e flessibile da permettere la realizzazione tramite esso anche di altri meccanismi di rappresentazione ed elaborazione della conoscenza, nonche' di vari strumenti tipici dei sistemi esperti, quali meccanismi di spiegazione e di ragionamento incerto [2]. In questo modo pero' si viene a mescolare conoscenza sul dominio da modellare con quella di controllo, e cioe' con predicati e clausole particolari aggiunti per guidare l'interprete Prolog nel modo desiderato. I programmi risultano percio' poco comprensibili e difficilmente modificabili.

Questi problemi possono essere in parte superati dagli altri due approcci, che risultano pero' insoddisfacenti per altri aspetti, come la scarsa efficienza di esecuzione nell'uso indiretto del Prolog e la difficolta' di realizzazione nel caso di estensioni.

Si e' percio' deciso di indagare una tecnica diversa da quelle sopra citate, basata sull'uso di strumenti di 'preprocessing' [3]. In questo nuovo approccio, i meccanismi di descrizione ed interpretazione della conoscenza vengono ancora implementati in Prolog, ma i relativi programmi vengono generati automaticamente tramite preprocessing di una specifica di piu' alto livello, fornita dall'utente, contenente esclusivamente informazioni sulla conoscenza del dominio in oggetto. Tutte le informazioni sulla strategia di controllo vengono cosi' inserite dal preprocessor in termini di opportune clausole e predicati Prolog e sono percio' nascosti all'utente stesso. Il preprocessor a sua volta puo' essere facilmente realizzato in Prolog.

Tale metodo e' stato utilizzato in particolare per costruire schemi di rappresentazione basati su regole di produzione (capitolo 2) e su un paradigma che combina frames e regole (capitolo 3).

2. REGOLE DI PRODUZIONE

Il primo passo nella definizione di schemi per la rappresentazione della conoscenza e' stata la definizione di alcuni semplici sistemi a regole di produzione utilizzanti strategie di inferenza diverse da quella backward offerta dal Prolog [3] [4]. In particolare, sono state implementate le seguenti strategie:

- inferenza forward
- inferenza backward con incertezza
- inferenza forward con incertezza.

Si consideri ad esempio il problema di effettuare inferenza 'forward' (ossia guidata dai dati) sul seguente semplice sistema di regole:

```
a1 if b1 or b2.
a2 if a3 and b4.
a3 if b3 and a1.
goal(a2).
```

Figura 1.

Almeno un fatto del tipo goal(A) deve apparire nel sistema di regole a

indicare che A e' un possibile obiettivo ('goal') della deduzione.

Le clausole Prolog generate nella fase di 'preprocessing' (contenenti quindi predicati 'di controllo') che consentono la deduzione guidata dai fatti sono descritte in Figura 2.

```
start :- a2,!,write(a2).
start :- rule1.
start :- rule2.
start :- rule3.
start :- write('nessun goal dimostrabile').

rule1 :- not(a1),
         ( b1 ; b2 ),
         insert(a1),
         start.
rule2 :- not(a2),
         a3,b4,
         insert(a2),
         start.
rule3 :- not(a3),
         b3,a1,
         insert(a3),
         start.
```

Figura 2.

I fatti iniziali devono essere forniti nella forma di usuali asserzioni Prolog. Ad esempio, b1, b3, b4 e' un possibile insieme di dati iniziali per il sistema di Figura 1. L'esecuzione e' attivata quando il goal 'start' e' fornito all'interprete Prolog. La prima clausola per 'start' contiene l'obiettivo del sistema; se tale obiettivo e' soddisfatto l'esecuzione termina con successo. Altrimenti le alternative successive per 'start' sono prese in considerazione, portando cosi' alla attivazione della prima regola il cui antecedente e' soddisfatto e che aggiunge nuova conoscenza all'insieme dei fatti noti. L'esecuzione termina quando, dopo un certo numero di passi di deduzione, un obiettivo del sistema di regole e' dimostrato o nessun obiettivo e' dimostrato, ma nessuna regola e' piu' attivabile.

Il predicato 'insert' e' usato al posto del predicato di sistema 'assert' ed e' definito come segue:

```
insert(C):- assert(C).
insert(C):- retract(C),fail.
```

Ogni volta che la clausola C e' inserita nel database, una alternativa nell'albero di esecuzione del Prolog per rimuovere C e' mantenuta aperta; tale alternativa verra' selezionata in caso di backtracking su 'insert'.

L'esempio di deduzione 'forward' proposto e' molto semplice. In modo simile e' stato implementato anche uno schema di deduzione piu' complesso che consente di selezionare ad ogni passo la regola 'migliore' (in base a cosiderazioni euristiche) da mandare in esecuzione (nel programma di Figura 2 invece la prima regola attivabile e' mandata subito in esecuzione). In tale

caso ad ogni passo, corrispondente ad una invocazione di 'start', si distinguono due fasi:
- match, per collezionare nel 'conflict set' le regole che risultano soddisfatte.
- risoluzione dei conflitti, per selezionare ed attivare una regola del "conflict set". La strategia di selezione puo' essere basata su considerazioni euristiche generali o dipendenti dal dominio di applicazione.

In questo modo l'efficienza della deduzione forward puo' essere notevolmente migliorata, focalizzando il processo di ricerca.

Sono stati anche sviluppati semplici meccanismi di spiegazione e di debugging per i sistemi a regole di produzione realizzati, inserendo opportuni predicati di controllo nel codice Prolog generato dal preprocessor.

Una descrizione piu' precisa ed esempi di sistemi di regole con deduzione con incertezza sono riportati in [3] e [4].

3. FRAMES

Il passo successivo e' stato quello di applicare le stesse tecniche alla implementazione di schemi di rappresentazione a 'frame' [5]. Piu' in particolare si e' rivolto l'interesse verso schemi di rappresentazione misti basati sulla combinazione a vari livelli di frame e regole di produzione, suggeriti dalle necessita' di rappresentazione in vari domini reali (domini medici in particolare; cfr. ad es. [6]). L'obiettivo fondamentale che ci si e' preposti e' stato quello di ottenere una rappresentazione il piu' possibile flessibile e adatta quindi a diversi domini di conoscenza [6].

L'organizzazione complessiva del "frame-system" e' di tipo gerarchico, con la possibilita' di specificare non solo legami di specializzazione, ma anche di suggerimento di ipotesi alternative. Il livello piu' alto della gerarchia e' occupato da un frame particolare, detto superframe, contenente conoscenza indipendente dal dominio (conoscenza sul controllo), il cui compito e' essenzialmente quello di gestire la consultazione iniziale dell'utente e di organizzare ad alto livello la strategia di ricerca, secondo un meccanismo di "agenda" [7]. Il superframe e' generato automaticamente in fase di preprocessing ed e' quindi trasparente all'utente.

Ogni frame contenente conoscenza del dominio (definito dal progettista della conoscenza) puo' avere diverse forme a seconda delle esigenze espresse dall'utente nella descrizione ad alto livello del sistema. Piu' precisamente e' possibile avere:
- frame contenenti sistemi a regole di produzione con interpretazione sia forward che backward (con o senza incertezza), costruiti con la tacnica descritta nel capitolo precedente;
- frame con slot sia in and che in or. Nel caso di slot in or e' sufficiente l'istanziazione di uno slot per ottenere quella del frame; nel caso di slot in and invece e' necessario che vengano istanziati tutti gli slot.

L'organizzazione generale di un frame e' mostrata in Figura 3.

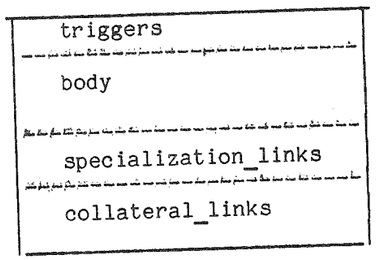


Figura 3.

In tale schema si riconoscono le quattro componenti fondamentali di un frame:
- Triggers, o condizioni di triggering, unica parte di un frame visibile da altri frames, che consentono di stabilire se sia o no sensato, in un certo contesto, attivare il frame.
- Body, parte centrale del frame che, come accennato, puo' essere costituita da un sistema di regole di produzione (nel qual caso attivare il frame coincide con l'attivare il sistema di regole) o da un insieme di slot.
- Legami di specializzazione, che connettono un frame ai suoi sottoframes (che definiscono specializzazioni nell'organizzazione gerarchica). Tali legami sono utilizzati nella fase di ricerca dopo la istanziazione del frame, per particolareggiare la soluzione da esso proposta.
- Legami collaterali, utilizzati invece nel caso in cui la istanziazione del frame fallisca, per suggerire ad un suo antenato nella gerarchia o al superframe altri frames, ossia altre ipotesi, da prendere in considerazione nel processo di ricerca.

Nel caso di frame basati su slot, l'organizzazione degli slot risulta particolarmente flessibile. Uno slot puo' essere istanziato in tre modi possibili:
- verifiche di fatti;
- domande all'utente;
- attivazione di eventuali regole di produzione associate allo slot;
- attivazione di un programma Prolog associato allo slot (attachment procedurale).

Piu' modi diversi possono essere usati congiuntamente. Si possono anche definire piu' tentativi di istanziazione alternativi, con modalita' differenti.

La strategia di ricerca del sistema e' molto semplice. In primo luogo viene attivato il superframe che, dopo aver controllato la fase di consultazione, costruisce, usando i triggers dei suoi sottoframes, l'agenda iniziale delle ipotesi da prendere in considerazione. Ha quindi inizio il ciclo di ricerca di una soluzione. Ad ogni passo, ossia ogni volta che il controllo ritorna al superframe, l'ipotesi piu' plausibile tra quelle contenute nell'agenda, viene selezionata ed il corrispondente frame attivato. Ha cosi' inizio il tentativo di istanziazione di tale frame e di sue eventuali specializzazioni. Se tale tentativo ha successo, viene sintetizzata una

spiegazione della soluzione e di come e' stata determinata, spiegazione che verra' proposta all'utente, e il ciclo di ricerca termina; altrimenti nuove ipotesi da prendere in considerazione vengono suggerite al superframe attraverso i legami collaterali dei frames non istanziati e inserite nell'agenda. Si noti ancora che e' possibile ottenere dal sistema una ricerca esaustiva di tutte le possibili soluzioni ad un problema: l'utente ha infatti la possibilita' di richiedere la prosecuzione della ricerca ogni volta che una soluzione gli viene proposta.

Per quanto riguarda l'implementazione, si e' seguito l'approccio descritto in precedenza, basato sull'uso di un preprocessor. Come primo passo sono stati percio' sviluppati i programmi Prolog in grado di realizzare il sistema a frame descritto. Successivamente, e' stato definito un linguaggio per la descrizione della conoscenza, tale da permettere all'utente di astrarre dai dettagli della implementazione Prolog. Il linguaggio ha una sintassi tipo Prolog e risulta di facile utilizzo. Infine, e' stato definito e completamente implementato in Prolog, il programma traduttore (preprocessor) per tale linguaggio. Quindi, allo scopo di provare l'effettiva usabilita' del sistema a frame, e' stato sviluppato un semplice sistema esperto per la diagnosi di guasti meccanici.

4. CONCLUSIONI E SVILUPPI FUTURI

Il metodo utilizzato presenta notevoli vantaggi sia in assoluto che rispetto ad altri usi del Prolog per la costruzione di sistemi esperti, tra cui principalmente:
- possibilita' di fondere insieme in modo naturale piu' formalismi di rappresentazione e meccanismi di inferenza;
- semplicita' di utilizzazione;
- possibilita' di sfruttare al massimo le caratteristiche del Prolog.

Due differenti linee di sviluppo sono attualmente allo studio, a partire dai risultati descritti in precedenza.
Da una parte si sta cercando di migliorare il sistema di rappresentazione sino ad ora costruito, potenziandone la capacita' rappresentativa per renderlo piu' adeguato per realizzazioni in domini specifici. In particolare si sta tentando di disegnare uno schema di rappresentazione della conscenza a piu' livelli, i cui si riconosca:
- conoscenza superficiale, descritta usando l'attuale schema basato su frames e regole di produzione
- conoscenza profonda o causale [8] [9], descritta usando un particolare schema basato su reti causali.

I due livelli sono integrati, con diversi passaggi dall'una all'altra forma di ragionamento.

D'altra parte e' allo studio lo sviluppo in Prolog di un diverso meccanismo di rappresentazione ed elaborazione della conoscenza, basato sul modello ad oggetti. Come primo risultato di questo lavoro sono state aggiunte un certo numero di funzionalita' al Prolog per permettere essenzialmente la definizione di un oggetto con il suo insieme di metodi associati, la definizione, ricerca e applicazione di metodi, lo scambio di messaggi e l'ereditarieta' tra classi. L'approccio seguito in questo caso e' stato

quello di un semplice uso diretto del Prolog, con la definizione di un certo numero di predicati in grado di supportare le operazioni sugli oggetti sopra citate. Quanto sviluppato finora costituisce una semplice estensione del lavoro di Zaniolo sulla programmazione ad oggetti in Prolog [10].

Sulla base di questa prima esperienza, si sta anche sviluppando un sistema piu' complesso e completo in grado di integrare i due paradigmi di programmazione, quello ad oggetti e quello logico. L'approccio che si sta seguendo e' simile a quello proposto in [11] e si prevede lo sviluppo di un livello di interpretazione/compilazione da aggiungere sopra al Prolog, nonche' eventualmente limitate modifiche all'interprete Prolog stesso. In prospettiva poi si prevede anche una possibile integrazione di questo sistema con quello a frame sopra descritto per giungere ad un sistema di elaborazione della conoscenza piu' completo e sofisticato.

5. RIFERIMENTI BIBLIOGRAFICI

[1] Rossi, G.: Uses of Prolog in Implentation of Expert Systems; sottomesso per pubblicazione, Giugno 1985.

[2] Clark, K.L. and McCabe, F.G.: PROLOG; A Language for Implementing Expert Systems, Machine Intelligence, 10 (Hayes & Michie eds.), 1982.

[3] L.Console, G.Rossi: Implementing Inference Strategies in Prolog by Preprocessing; Rapporto Interno, Dip. di Informatica, Univ. di Torino, Aprile 1985.

[4] L.Console, G.Rossi: Implementing Inference Strategies in Prolog based Expert Systems; to be presented at the 8th European Meeting on Cybernetics and System Research, Wien 1986.

[5] Minski, M.: A framework for representing knowledge; in The psychology of computer vision (P. Winston ed.) Mc Graw Hill 1975.

[6] L.Console, G.Rossi: Frame Based Expert Systems in Prolog; Rapporto Interno, Dip. di Informatica, Univ. di Torino, Novembre 1985.

[7] Aikins, J.: Prototypical knowledge for expert systems; in Artificial Intelligence 20, 1983. pp 163,210

[8] Patil, R: Causal representation of patient illness for Electrolyte and Acid-Base diagnosis; Ph. D. Th. M.I.T., MIT/LCS/TR-267, October 1981.

[9] Long, W.: Causal Reasoning in a physiological model as a Computational Paradigm; Proceedings of IEEE, MEDCOMP Conference 1983.

[10] C.Zaniolo: Object-oriented programming in Prolog; Proc. of the International Symposium on Logic Programming, Atlantic City, N.J., February 6-9, 1984, 265-270.

[11] M.Tokoro, Y.Ishikawa: An Object-Oriented Approch to Knowledge Systems; Proc.of the Int. Conf. on 5th. Generation Computer Systems 1984, Tokyo, Japan, Nov. 6-9, 1984, 623-631.

REALIZZAZIONE DI UN SISTEMA DI RAPPRESENTAZIONE DELLA CONOSCENZA IN AMBIENTE PROLOG

Riccardo Bisi – Marco Boero

Dipartimento di Informatica, Sistemistica e Telematica
Universita` di Genova, Via Opera Pia 11A, 16145 Genova

SOMMARIO

Si presentano alcune ipotesi relative alla realizzazione di un sistema di rappresentazione della conoscenza nell'ambito della programmazione logica. In particolare e` affrontata l'integrazione dei meccanismi "strutturali" (reti semantiche,frames) in ambiente PROLOG. Nel sistema proposto, il deduttore PROLOG interagisce, oltre che con l'usuale insieme di clausole, con una tassonomia di termini, basata sulle nozioni dell'"ereditarieta` strutturata" sviluppate nel KL – ONE. E` descritta brevemente la funzionalita` dell'ambiente di rappresentazione, la realizzazione logica delle nozioni strutturali (ereditarieta`,classificazione), e l'interazione con l'interprete PROLOG.

1. Introduzione

Un aspetto rilevante nella rappresentazione della conoscenza e` la distinzione tra conoscenza descrittiva (o definitoria) – usualmente espressa come una tassonomia che classifica gli elementi del dominio – e conoscenza asserzionale – consistente nell'espressione di fatti relativi al mondo reale – [13,2].

Il confine tra i due tipi di conoscenza e` spesso incerto e, in genere, la maggior parte dei formalismi sino ad ora proposti non consente di rendere la differenza nel modo opportuno. Ad esempio, in sistemi basati sulla logica del prim'ordine, le definizioni non risultano distinguibili da asserzioni espresse sotto forma di bicondizionali. Nei linguaggi basati sui frames invece (ad es. KRL [12], FRL [10] e PEARL [1]), la conoscenza descrittiva e` incorporata nelle relazioni della tassonomia, mentre la conoscenza asserzionale e` resa unicamente attraverso l'istanziazione (l'inadeguatezza di questa soluzione e` discussa in [7]).

Un trattamento piu` soddisfacente di questo aspetto e` stato proposto recentemente. Il KL – ONE [11] e il KRYPTON [7] riservano infatti due formalismi distinti ai due tipi di conoscenza, nell'ambito dello stesso sistema di rappresentazione.

Il modello che proponiamo, si basa sull'integrazione del linguaggio descrittivo del KL – ONE (SI – net) nell'ambiente PROLOG, e porta (come nel KRYPTON), a una divisione della rappresentazione. I due tipi di conoscenza restano concettualmente separati, ma integrati in un unico ambiente logico.

2. Un modello di integrazione

Nel sistema considerato, la funzionalita` rappresentazionale e` ripartita tra due componenti : (i) una base di conoscenze strutturata come un SI – net (KBs), realizzata da un insieme di clausole, per la conoscenza descrittiva, e (ii) una base di conoscenze clausale (KBc), per la conoscenza asserzionale.Entrambi i sistemi di conoscenze sono utilizzati dal deduttore, quando il sistema e` interrogato. Nella costruzione di KB, e` possibile interagire con KBs e KBc, aggiungendo conoscenza fattuale (clausole PROLOG) o conoscenza descrittiva (elementi della rete strutturale). L'accesso alla rete e` ottenuto attraverso un insieme di primitive che consentono costruzione e mantenimento della base di conoscenze al livello di astrazione proprio della conoscenza strutturata (implementazione PROLOG delle funzioni descritte in [9]).

La struttura funzionale complessiva alla quale facciamo riferimento e` indicata in fig.1

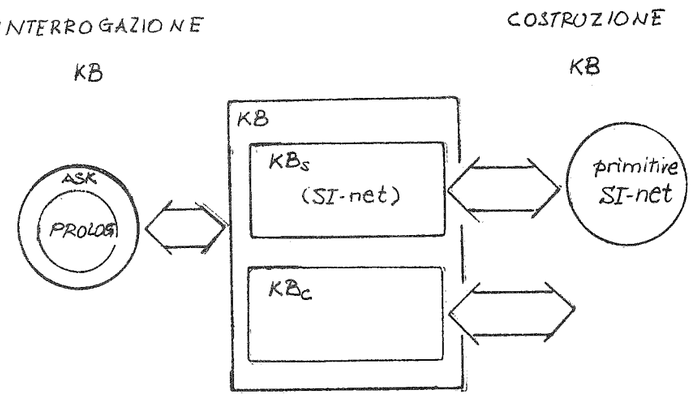


fig.1

KBs contiene una descrizione delle entita` concettuali che strutturano il dominio, mettendo in evidenza le relazioni tassonomiche (Super_c, Diff ecc...) e partonomiche ("ruoli", nodi quadri) tra i diversi tipi definiti. Ad esempio, potrebbe contenere la rappresentazione dell'SI – net mostrata in fig.2, che esprime una descrizione (parziale) dei concetti "stanza" e "ufficio"

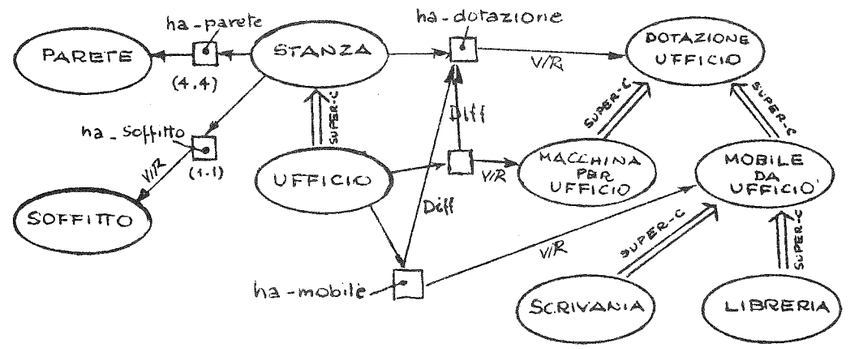


fig. 2

costruita con la sequenza di espressioni

```
assume(generic_concept(stanza))
add(stanza,generic_role,ha_parete)
add(ha_parete(stanza),vr,parete)
assume(generic_concept(ufficio))
assume(super_c(ufficio,stanza))
  ecc...
```

KBc potrebbe invece contenere fatti particolari come

```
ufficio(mio_ufficio).
scrivania(mia_scrivania).
ha_mobile(mio_ufficio,mia_scrivania).
```

e asserzioni generali come

```
nuovo(X): – ha_mobile(mio_ufficio,X),
not scaffale(X).
```

Le strutture di KBs sono interpretate come 'postulati di significato' che legano tra loro i predicati utilizzati per esprimere asserzioni sul dominio. Dall'insieme delle conoscenze definite in KBc e KBs e` allora possibile trarre le seguenti inferenze :

```
? – stanza(X).
      X = mio_ufficio
? – ha_dotazione(mio_ufficio,X).
      X = mia_scrivania
? – mobile(X),nuovo(X).
      X = mia_scrivania
```

3. La base della conoscenza strutturale

Una prima soluzione, per ottenere una rappresentazione degli oggetti dell'SI – net in termini di clausole PROLOG, puo` essere basata sulla nota considerazione che i linguaggi dei frames e delle reti semantiche sono in linea di massima traducibili in linguaggi del prim'ordine (associando predicati unari a frames e nodi, e predicati binari a slots o archi che rappresentano attributi [4]).

In questo modo, la rete di fig.2 puo` essere rappresentata con un insieme di clausole PROLOG (ricorrendo alla skolemizzazione delle variabili), espresse nel linguaggio oggetto che descrive il dominio :

```
stanza(X): – ufficio(X).
ha__soffitto(X,soffitto(X)): – stanza(X).
soffitto(soffitto(X)): – stanza(X).
    ecc...
```

Questa rappresentazione, con l'apparato deduttivo del PROLOG, consente di rendere la funzionalita` base della conoscenza strutturata (ereditarieta`,classificazione). Dalle espressioni quantificate, ad esempio, e` possibile derivare i seguenti fatti :

```
? – stanza(X)
      X = mio__ufficio
? – ha__sofffitto(X)
      X = mio__ufficio
      Y = soffitto(mio__ufficio)
? – soffitto(X)
      X = soffitto(mio__ufficio)
```

Con questa soluzione, la struttura della rete potrebbe essere utilizzata come una sorta di "linguaggio superficiale" per l'interazione, che permette all'utente di esprimere in termini piu` intuitivi la struttura della sua conoscenza descrittiva, mantenendo la logica dei predicati come linguaggio di rappresentazione.

Una limitazione fondamentale a questo tipo di soluzione risulta dalle caratteristiche espressive dell'SI – net, alcune delle quali appaiono difficilmente riducibili in forma logica, con i vincoli imposti dalle clausole di Horn (ad esempio, le restrizioni sulla "modalita`" e sulla "cardinalita`" dei ruoli).

Una soluzione attuabile, e` invece basata su una rappresentazione clausale della rete ad un meta – livello descrittivo; caratterizzando cioe`, in forma logica, la struttura della conoscenza e la funzionalita` relativa. Una rappresentazione di questo tipo e` costituita da :

a) una descrizione degli oggetti formali che definiscono l'organizzazione della conoscenza (concetti,ruoli,relazioni tassonomiche,ecc.),

b) un insieme di regole da usare per le inferenze logiche, che definiscono in forma deduttiva l'"ereditarieta` strutturata",

c) un insieme di regole che esprimono vincoli di consistenza, per garantire l'integrita` strutturale della rete nelle operazioni di acquisizione e modifica della conoscenza.

Ad esempio, le regole che esprimono la discendenza tra concetti e l'ereditarieta`, sono clausole PROLOG della forma :


```
super__c(X,Y): – super__c(link(X,Y)).
super__c(X,Y): – super__c(link(X,Z)),
                 Z =/= Y,
                 super__c(Z,Y).

individuates(X,Y): – individuates(link(X,Y)).
individuates(X,Y): – individuates(link(X,Z)),
                     Z =/= Y,
                     super__c(Z,Y).

has__role(X,R): – has__role(link(X,R)).
has__role(X,R): – has__role(link(Y,R)),
                  super__c(X,Y),
                  not (role(Q),Q =/= R,
                       super__role(Q,R),
                       has__role(X,Q)).
  ecc ...
```

I vincoli di consistenza della rete sono invece espressi come assiomi negativi, che in forma clausale sono scritti :

```
(1) ← super__c(X,X).
(2) ← super__c(X,Y),super__c(Y,X).
(3) ← has__role(link(X,R)),has__role(link(Y,R)),Y =/= X.
```

Queste clausole possono essere date al sistema di deduzione PROLOG come goals da soddisfare. Se qualcuno di essi ha successo si ha una violazione di consistenza. Se, ad esempio, il fatto has__role(link(Concept1,Role1)) e` gia`contenuto in KBs, il vincolo (3) impedisce l'aggiunta del fatto has__role(link(Concept2,Role1)).

Un tale modo di trattare la conoscenza negativa deriva da una implicita "closed world assumption" (CWA), usuale per le basi di dati caratterizzate in forma logica [8]. L'ipotesi CWA, unitamente ad altre assunzioni ("chiusura del dominio" e "unicita` dei nomi"), costituisce la parte implicita della rappresentazione, che e` realizzata, come solitamente avviene in PROLOG, attraverso l'interpretazione della negazione in termini di non – provabilita`.

4. Interrogazione della base delle conoscenze.

In PROLOG, e` agevole estendere il comportamento inferenziale dell'interprete logico, ereditandone direttamente la funzionalita` predefinita (backtracking e unificazione) [3].

Si puo` infatti introdurre un meta__livello di controllo che agisce come filtro per i goals da valutare, semplicemente partendo dal filtro ad effetto nullo

```
prove(true): – !.
prove((,(Goal1,Goal2))): – !,prove(Goal1),prove(Goal2).
prove(Goal): – clause(Goal,Body),prove(Body).
prove(Goal): – call(Goal).
```

ed estendendo il controllo, nel nostro caso, in modo da consentire l'accesso alla rete.

Cio` che si vuole ottenere e` un ciclo di valutazione che, dapprima, ricorre a KBc per soddisfare un goal, e successivamente, se fallisce, ne tenta il soddisfacimento attraverso KBs. L'accesso a KBs puo` generare ulteriori goals da soddisfare, attraverso un analogo ciclo, in modo che la prova, nel complesso, avviene con una serie di "riflessioni" tra i due livelli (clausole al livello oggetto e clausole che rappresentano la rete). Un comportamento di questo tipo e` ottenibile con la seguente

definizione dell'interrogazione :

```
ask(Query): – prove(Query).

prove(true): – !.
prove((,(Goal1,Goal2)))
        : – !,prove(Goal1),
          prove(Goal2).
prove(Goal): – sys__goal(Goal), !,  /* predicato
          sys__eval(Goal).          built – in
                                    PROLOG */
prove(Goal): – clause(Goal,Body),
          prove(Body).
prove(Goal): – klone__prove(Goal).
```

dove la procedura 'klone__prove' realizza l'accesso alla conoscenza strutturale, attraverso le primitive di interrogazione dell'SI – net.

In questo modo, se un goal non puo` essere soddisfatto attraverso KBc, il corrispondente predicato "p" e` interpretato per mezzo della classificazione dei tipi contenuta in KBs. Ad esempio interpretando la relazione di classificazione nel modo usuale

$$(\text{super\_\_c}(q,p) \quad \text{iff} \quad \forall(X,q(X) \supset p(X))),$$

la rete e` attraversata discendendo i "cables" che connettono i sottotipi a "p", generando i goals p1(X),...,pn(X) corrispondenti ai nodi raggiunti attraverso la gerarchia super__c. A ciascuno di essi e` ricorsivamente applicato il ciclo di prova definito.

Possiamo considerare alcune inferenze relative alla rete mostrata in fig.3, che supponiamo rappresentata in KBs.

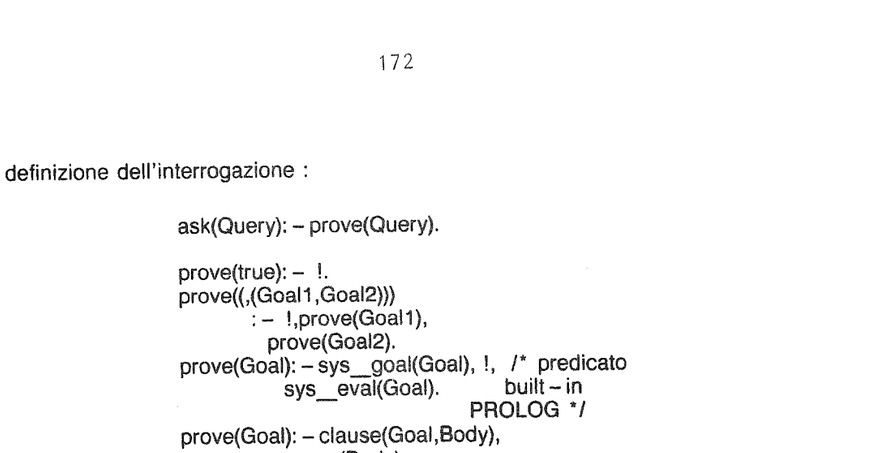


fig.3

Se KBc contiene i fatti

```
ufficio(mio__ufficio).
scrivania(mia__scrivania).
ha__mobile(mio__ufficio,mia__scrivania).
```

si possono trarre le seguenti inferenze :


```
stanza(mio__ufficio).
mobile(mia__scrivania).
mobile__da__ufficio(mia__scrivania).
```

in modo consistente alla classificazione dei concetti e alle restrizioni espresse dalla rete.

Dai fatti

```
ha__cassetto(mia__scrivania,cassetto1).
ha__cassetto(mia__scrivania,cassetto2).
ha__piano(mia__scrivania,piano__mia__scrivania).
cassetto(cassetto1).
```

inoltre, la formula

$$\text{ha\_\_parte}(\text{mia\_\_scrivania},X)$$

e` soddisfatta con le sostituzioni

$$X = \text{cassetto1} \ , \ X = \text{cassetto2} \ , \ X = \text{piano\_\_mia\_\_scrivania}$$

consistentemente alla classificazione dei ruoli (relazione di "Differenziazione"), mentre dall'ereditarieta` di un ruolo necessario (restrizione di "Modalita`") si puo` inferire

$$\text{ha\_\_peso}(\text{cassetto1})$$

Ad esempio, si puo` vedere come i due livelli di rappresentazione interagiscono nella prova del goal 'mobile(X)', coinvolgendo il valore di restrizione (V/R) di un ruolo :

```
klone__prove(mobile(X)) < – – > ? generic__concept(mobile)
                                has__vr(R,mobile)
                                  R = ha__mobile
                                has__role(W,R)
                                  W = stanza
                                univ(R,Y,X,G1)
                                 G1 = ha__mobile(X,Y)
                                prove(G1)
                                  X = mia__scrivania
                                  Y = mio__ufficio
                                univ(W,Y,G2)
                                 G2 = stanza(mio__ufficio)
                                prove(G2)
      yes
```

In questo caso, i termini 'mio__ufficio' e 'mia__scrivania' nell'asserzione 'ha__mobile(mio__ufficio,mia__scrivania)' ereditano le restrizioni espresse dalla conoscenza generale rappresentata nella rete.

5. Conclusioni

Abbiamo analizzato brevemente l'integrazione di un formalismo di strutturazione della conoscenza nell'ambiente operativo del PROLOG, e un'ipotesi di interazione tra la rappresentazione strutturata e la conoscenza espressa sotto forma di clausole logiche.

Quanto proposto costituisce un'analisi ad un livello appena preliminare, e dovrebbe servire come punto di partenza per l'approfondimento delle problematiche

sul tema.

Tra i punti che appaiono sin d'ora richiedere ulteriore attenzione includiamo un'indagine piu` approfondita dell'interazione tra i due livelli di rappresentazione, e un'estensione dell'espressivita` del linguaggio asserzionale, con particolare riferimento alla possibilita` di ragionare con eguaglianza [5], e di trattare in modo esplicito "incompletezza" nella conoscenza costruita in KB [6].

6. Ringraziamenti

Ringraziamo G.Adorni, A.Cappelli, S.Gaglio, L.Moretti per averci seguito nello sviluppo di questo lavoro.

BIBLIOGRAFIA

1. M.Deering, J.Faletti, R. Wilensky , "PEARL, a Package for Efficent Access to Representation in LISP," *Proc IJCAI – 1981*, pp. 930 – 932 (August,1981).

2. R.J. Brachman, "What's in a Concept: structural foundations for semantic networks," *International Journal of Man – Machine Studies,9*, pp. 127 – 152 (1977).

3. Jacques Cohen, "Describing Prolog by its interpretation and compilation," *CACM* 28(12) pp. 1311 – 1324 (Dec. 1985).

4. P.J. Hayes, "The Logic of Frames," pp. 46 – 61 in *Frame Conceptions and Text Understanding*, ed. D.Metzing,Walter de Gruyter and Co., Berlin (1979).

5. W. Kornfeld, "Prolog with equality," *IJCAI – 83*, pp. 514 – 519 (August 83).

6. Hector J. Levesque, "The interaction with Incomplete Knowledge Bases : a Formal Treatment," *IJCAI – 81*, pp. 240 – 245 (August 1981).

7. Ronald J. Brachman, Richard E. Fikes,Hector J. Levesque, "KRYPTON: A Functional Approach to Knowledge Representation," FLAIR Technical Report No.16 (May, 1983).

8. Herve` Gallaire, Jack Minker, Jean – Marie Nicolas, "Logic and Databases: A Deductive Approach," *ACM Computing Surveys* 16(2) pp. 153 – 185 (June 1984).

9. R.Brachman,E.Ciccarelli,N.Greenfeld,M.Yonke, "KL – ONE Reference Manual," BBN, Tech.Rep. No.3848, Cambridge, Massachusetts (July 1978).

10. Ira P. Goldstein, R. Bruce Roberts, "NUDGE , a knowledge – based scheduling program," *IJCAI – 77*, pp. 257 – 263 (August 1977).

11. Ronald J. Brachman, James G. Schmolze, "An Overview of the KL – ONE Knowledge Representation System," *Cognitive Science, 9*, pp. 171 – 216 (1985).

12. Daniel G. Bobrow, Terry Winograd, "An overview of KRL, a knowledge representation language," *Cognitive Science, 1:1*, (1977).

13. W. Woods, "What's in a Link: Foundations for Semantic Networks," in *Representation and Understanding*, ed. D. Bobrow, A. Collins,Academic Press (1975).

Un sistema esperto per l'elaborazione di segnali

Patrizia Rossi, Roberto Chicchiero

Selenia S.p.A.

ABSTRACT

Un sistema per l'elaborazione di segnali radar riconosce emittenti radar a partire da singoli impulsi rilevati nell'ambiente circostante. I vantaggi offerti dalle tecnologie expert system e dai linguaggi a regole per realizzare tali sistemi, fanno sì che molti difetti degli attuali riconoscitori vengano eliminati. Uno dei vantaggi dei linguaggi a regole rispetto ai linguaggi imperativi è una maggiore semplicità nel modificare ed incrementare i programmi, caratteristica molto utile nei sistemi di riconoscimento radar. E' stato scelto il Prolog come linguaggio di implementazione poiché, rispetto agli altri linguaggi a regole, offre il vantaggio di una semantica chiara e semplice.
Il lavoro comprende inoltre l'implementazione di una interfaccia funzionale da Prolog a C.

1. Introduzione

Un sistema per riconoscimento elettronico radar [4] è un sistema che analizza automaticamente l'ambiente elettromagnetico circostante con lo scopo di rivelare e caratterizzare le emissioni radar presenti. Questi sistemi sono usati in campo militare per distinguere emittenti alle quali può essere associato un sistema d'arma dalle emittenti usate per la normale navigazione aerea o marittima.
Questi sistemi hanno bisogno di frequenti modifiche e aggiunte di nuovi programmi o dati a causa del rapido evolversi delle situazioni sul campo; inoltre deve essere possibile provare la validità delle modifiche da apportare prima di inserirle nel sistema.
Per questi motivi è utile lavorare prima su un prototipo del sistema che anche se non può essere utilizzato in tempo reale consente di provare tali modifiche.
La nostra proposta è di utilizzare il Prolog [1,3] per scrivere il prototipo del sistema perché soddisfa queste esigenze; inoltre essendo un linguaggio a regole permette di inserire conoscenza in modo non vincolato dal controllo della computazione, fatto che semplifica la scrittura di programmi che si comportano come esperti.
La problematica del riconoscimento elettronico richiede la codifica della conoscenza di un esperto di apparati radar, oltre ad algoritmi ben determinati, perché si presentano anomalie nella ricezione che rendono difficile il riconoscimento. Le irregolarità nella ricezione dei segnali possono dipendere dalle emittenti, che non vogliono farsi individuare, dal ricevitore stesso, che ha dei propri limiti fisici, o dal mezzo di trasmissione che può causare eventi inaspettati.
La prima parte del nostro sistema è di natura procedurale e perciò è stata realizzata in C, mentre il resto è implementato in Prolog. Per riuscire a fondere i diversi tipi di conoscenza, procedurale e dichiarativa, abbiamo realizzato all'interno dell'interprete C-Prolog un "functional attachment" che permette di eseguire funzioni C nell'ambito di goal Prolog.

2. Sistemi per riconoscimento elettronico

Un sistema che elabora segnali radar deve essere in grado, data una sequenza temporale di impulsi provenienti da diverse emittenti, di ricostruire sequenze provenienti dalla stessa emittente. Perché il riconoscimento abbia successo un sistema di questo tipo deve contenere la conoscenza riguardante i diversi modi di trasmissione delle varie emittenti.

I dati relativi ad ogni singolo impulso riguardano:

a) frequenza di trasmissione
b) tempo di arrivo
c) direzione di arrivo
d) ampiezza
e) durata.

Un sistema per il riconoscimento elettronico può essere diviso a livello logico in due parti: la prima compie operazioni matematiche sugli impulsi per formulare ipotesi sulle possibili emittenti; la seconda valuta le ipotesi basandosi principalmente sulla conoscenza dell'esperto in apparati radar. Nella prima fase il sistema prende i dati da un ricevitore, che misura i parametri di ogni segnale captato, e li elabora in modo da poter formulare ipotesi sulle emittenti che hanno generato i segnali. Per riconoscere le emittenti a partire da singoli impulsi è necessario ricostruire sequenze di impulsi le cui caratteristiche siano note; si confrontano i parametri più significativi degli impulsi (frequenza, durata, direzione di provenienza) creando delle possibili sequenze in base al risultato dei confronti. Basandosi sulle sequenze così create è possibile calcolare altri parametri che caratterizzano le emittenti, ad esempio il periodo di ripetizione degli impulsi. Nella successiva fase di valutazione delle ipotesi è fondamentale la conoscenza dell'esperto di apparati radar, poiché la regolarità delle sequenze di impulsi generati dalle emittenti può essere alterata da eventi dipendenti dalle emittenti, dai limiti fisici del ricevitore e dal mezzo di trasmissione. La capacità di risolvere le difficoltà di riconoscimento legate a questi fenomeni rende il sistema "esperto".

3. Perché AI

Come abbiamo visto un sistema di riconoscimento elettronico può essere suddiviso a livello logico in due parti, di cui la prima compie operazioni piuttosto meccaniche e sequenziali, mentre la seconda è basata in gran parte sull'esperienza dei conoscitori di radar e perciò si presta meglio ad essere trattata con le tecniche degli expert systems [2].
Tuttavia le prestazioni richieste al nuovo sistema perché sia superiore a quelli attuali fanno sì che sia conveniente implementare con un linguaggio a regole, almeno in fase protipale, anche la prima parte.

Le caratteristiche richieste al nuovo sistema sono:

a) la soluzione di almeno una parte dei casi in cui il precedente tipo di sistemi falliva nel riconoscimento;
b) incrementalità;
c) modificabilità;
d) prototipazione e testing veloce.

Come si vede gli ultimi tre punti riguardano più il linguaggio usato per implementare il sistema che non il sistema vero e proprio; sostanzialmente, infatti, sono gli stessi requisiti richiesti in genere ad un linguaggio di specifica.

Ciò che abbiamo provato con questo lavoro è che il Prolog è un buon linguaggio di specifica con il vantaggio in più di essere eseguibile e quindi di poter eseguire direttamente le specifiche.

Il punto a), per il tipo di problemi che pone, è quello che caratterizzerà tutto il sistema come esperto. Come vedremo esso contiene alcune classi di eventi che rendono il problema del riconoscimento un problema per esperti.

4. Difficoltà nel riconoscimento

Il riconoscimento di un radar è reso difficoltoso da tre classi di eventi che modificano la regolarità della sequenza di impulsi ricevuti.

i) <u>eventi dipendenti dalle emittenti</u>
Le emittenti, specie nel caso siano associate ad un sistema d'arma, tendono volontariamente a rendere più difficile o più lento il riconoscimento. Questi eventi riguardano variazioni della frequenza di trasmissione degli impulsi e/o variazioni dell'intervallo di tempo da un impulso all'altro. Tali variazioni seguono leggi ben determinate o pseudocasuali.

ii) <u>eventi dipendenti dal ricevitore</u>
Le apparecchiature di ricezione hanno ovviamente dei limiti fisici, perciò eseguono le letture con un certo grado di errore, non ricevono impulsi sotto una certa soglia di potenza, non riescono a distinguere due impulsi che arrivano contemporaneamente da due emittenti diverse.

iii) <u>eventi dipendenti dal mezzo</u>
Il mezzo di trasmissione, l'atmosfera per esempio, introduce fenomeni fortemente variabili ed inaspettati quali riflessioni (su montagne, case o altro) o propagazioni anomale dovute a particolari condizioni. Si pensi che in certe condizioni, particolari strati atmosferici si possono comportare come guide d'onda.

5. Tipo di conoscenza

Come esempio del tipo di conoscenza esperta inglobata nel sistema consideriamo la soluzione dei problemi posti dall'alterazione dei segnali dovuta ad eventi dipendenti dal mezzo trasmissivo.
Una tipica regola si presenta nel seguente modo:

> *Se la fase 1 ha scoperto due radar in cui tutte le caratteristiche sono uguali tranne la direzione di arrivo, allora c'è un'alta probabilità che una delle due emittenti sia creata da un riflesso dell'altra su qualche ostacolo che si trova nelle vicinanze.*

Una successiva analisi del territorio permetterà di scoprire quale delle due è il riflesso e quale l'emittente vera.

Un altro esempio di regola è il seguente:

> *Se durante la ricostruzione di una sequenza di impulsi si presenta una irregoalrità nella distanza temporale da un impulso ad un altro, rispetto agli altri impulsi della sequenza, allora si devono analizzare le cause di questo evento.*

Una delle possibilità è che il ricevitore abbia perso alcuni impulsi, o perché sono giunti sovrapposti ad altri, oppure perché avevano una potenza troppo bassa o altro. Ciò che si deve fare in tal caso è vedere se inserendo qualche impulso ad hoc si riesce a ricostruire una sequenza con un alto grado di confidenza.

6. Perché il Prolog

Le richieste fatte al punto 3, in particolar modo i punti b,c,d, ci hanno fatto scegliere fin dall'inizio del progetto un linguaggio a regole. La scelta del Prolog tra i linguaggi a regole è venuta da considerazioni sulla pulizia e chiarezza semantica del Prolog [6] . La chiarezza e la semplicità della semantica del Prolog assicurano in ogni momento un alto grado di comprensibilità dei programmi ed aiutano anche in fase di debugging.
Una volta scelto il Prolog i requisiti b,c,d sono stati automaticamente soddisfatti. Vediamoli uno alla volta.

- incrementalità
  le attenzioni che bisogna avere per incrementare in modo corretto un programma Prolog sono molto limitate rispetto ad un qualsiasi linguaggio procedurale. Questo è dovuto più che altro al fatto che la regola di scoping delle variabili è unica ed è molto semplice, ma anche alla totale assenza di effetti laterali, se non si usano i predicati che modificano i programmi a run-time (assert e retract).

- modificabilità
  per la modificabilità valgono gli stessi discorsi fatti per l'incrementalità; in più si ha quasi totale assenza di costrutti del controllo, cosa che semplifica in modo ovvio le modifiche.

- prototipazione rapida
  questa è una caratteristica molto ricercata nei linguaggi, specie in quelli usati in ambito industriale. Il passaggio dalla specifica di un problema al programma Prolog che la implementa è quasi un'operazione compilativa; la ragione di questo è tutta da trovarsi nella quasi assenza di costrutti del controllo nel linguaggio. Il doversi occupare di "che cosa" invece che di "come" durante la programmazione, semplifica effettivamente e notevolmente la scrittura dei programmi.

La nostra proposta, in definitiva, sostiene l'uso del Prolog per specificare e risolvere tutta quella classe di problemi che possiamo definire "da expert system", quei problemi cioè che presentano dei punti in cui non si sa bene come fare per risolverli o che si presentano "difficili"; un esempio di questi problemi è appunto quello del riconoscimento elettronico.
Da notare che un riconoscitore elettronico gira in un tempo reale molto stretto e di conseguenza con le tecnologie attuali siamo ben lontani dal far funzionare un sistema scritto in Prolog con tali vincoli. La prototipazione in Prolog però ci fà capire esattamente la natura del problema e ci permette di sviscerarlo in tutte le situazioni che all'inizio non erano ben comprese. Questo fornisce la specifica, stavolta esatta, per un'implementazione in un linguaggio procedurale.

7. Interfaccia procedurale

L'attività in Selenia (Pomezia) per quanto riguarda AI si compone di due linee: una applicativa, che riguarda maggiormente la produzione di expert systems, e l'altra che definiamo sistemistica, stimolata da quella applicativa, in cui si costruiscono dei tools generali da usare per produrre expert systems : l'expert system builder. In quest'area è stato prodotto un sistema di spiegazioni per qualunque programma Prolog, un editore guidato dalla sintassi per il Prolog, ed ora si è prodotta un'interfaccia procedurale da Prolog a C.
L'esigenza di poter disporre di un'interfaccia procedurale è sorta durante l'analisi del problema del riconoscimento elettronico, poiché la prima parte del riconoscitore è composta principalmente di calcoli matematici sequenziali. Il Prolog non è molto adatto né molto efficiente per trattare la conoscenza di tipo matematico; per questo abbiamo pensato di scrivere un modulo che permettesse di chiamare dal Prolog procedure scritte in C.
Tutto ciò non ha la pretesa di essere una integrazione tra i due linguaggi, ma è solo un modo per poter scrivere tutto ciò che è inerentemente procedurale con una vera e propria procedura; inoltre la possibilità di chiamare procedure C nel corpo di procedure Prolog impone che i parametri formali delle procedure Prolog siano fissati come parametri di ingresso o di uscita una volta per tutte.

Le intestazioni di procedura scritte in C che verranno chiamate da Prolog devono contenere sia i parametri di ingresso che quelli di uscita e vengono chiamate come una normale procedura Prolog. Quando viene generato un fallimento dal mancato matching della chiamata C con le procedure Prolog, prima di attivare il backtracking, il meta-interprete Prolog attiva un processo che va ad eseguire la procedura, se esiste, scritta in C. L'esistenza è provata tramite una dichiarazione da fare in Prolog del tipo :

```
nonprolog(nome-file,nome-procedura).
```

L'interfaccia è stata realizzata tramite un meta- interprete scritto in Prolog che attiva l'interprete Prolog o l'esecutore C a seconda del goal che deve essere eseguito. Il passaggio dei parametri avviene tramite un pipe Unix.

Il Prolog è uno dei linguaggi che rende possibile scrivere un interprete di se stesso in se stesso, poiché non fà distinzione tra dati e programmi. Questa possibilità semplifica notevolmente la scrittura di interfacce procedurali come pure di moduli di spiegazione o di interfacce per acquisire conoscenza o altro [5].

8. Conclusioni

L'uso del Prolog nella progettazione di sistemi esperti permette in primo luogo la prototipazione veloce del sistema da realizzare, ed inoltre garantisce un discreto grado di comprensibilità, modificabilità ed aumentabilità dei programmi. Queste caratteristiche sono molto utili in tutti quei sistemi soggetti a frequenti modifiche a causa di condizioni variabili dell'ambiente circostante, come ad esempio i sistemi per riconoscimento elettronico.
Il Prolog si è dimostrato un buon linguaggio per definire la specifica del sistema esperto in riconoscimento elettronico di apparati radar da noi realizzato. Questa specifica consente una riprogrammazione del sistema esperto in un linguaggio più adatto ad elaborazioni in tempo reale richieste dalla problematica trattata.

Ringraziamenti

Ringraziamo il Prof. Franco Turini per la supervisione e l'organizzazione generale del lavoro. In modo particolare ringraziamo gli Ingg. Valter Marziali (Selenia Pomezia) e Sergio Pardini (Selenia Pisa) per aver svolto la fondamentale funzione di esperti di riconoscimento elettronico e di apparati radar.

Bibliografia

1. W.F. Clocksin, C.S. Mellish. **Programming in Prolog.** Springer-Verlag 1981
2. F. Hayes-Roth, D. Waterman, D. Lenat. **Building Expert Systems,**Addison Wesley 1983.
3. R.A. Kowalski. **Logic for Problem Solving.** North Holland 1979
4. V. Marziali. **La Logica di Estrazione dell'ATAD.** Rapporto Tecnico N.82108 1982 Selenia S.p.A.
5. L. Sterling. **Expert System = Knowledge + Meta-Interpreter.** In corso di pubblicazione
6. M.H. Van Emden, R.A. Kowalski. **The Semantics of Predicate Logic as a Programming Language.** Journal ACM V.23 N.4 1976

RAPPRESENTAZIONE DI TESTI DI LEGGE IN PROLOG

Roberto Bertocchi

Istituto di Cibernetica - Università degli Studi di Milano
via Viotti 5, 20133 Milano

ABSTRACT

In questo articolo viene descritta una formalizzazione in termini logici del contenuto di testi legislativi, sono discussi i problemi di rappresentazione della conoscenza coinvolti, ed é mostrato come la formalizzazione può essere utilizzata per supportare le attività regolamentate dalle norme di legge.
Il dominio legale è un ambito privilegiato per la studio di problemi di rappresentazione poiché la conoscenza è specificata in testi in linguaggio naturale.
La rappresentazione logica di norme legali chiarifica il contenuto dei testi legislativi e permette la costruzione di sistemi esperti di tipo legale.
Le tecniche e metodologie impiegate in quest'ambito possono essere utilizzate anche per lo sviluppo di sistemi software tradizionali, sistemi d'ufficio e sistemi di supporto alla decisione.

INTRODUZIONE

Gran parte dell'attività delle organizzazioni è regolamentata da norme legali, la rappresentazione di tali regole in una forma comprensibile da un elaboratore è quindi di estrema importanza.
La rappresentazione delle norme legali costituisce sia un metodo per chiarire e rendere comunicabile il contenuto delle leggi che la base per la realizzazione di sistemi che permettano di utilizzare praticamente tale conoscenza limitando il ricorso ad esperti umani.
Sistemi software che incorporano la conoscenza di una normativa possono applicare automaticamente la legge in specifiche circostanze o simularne l'applicazione, permettendo di identificare punti controversi e implicazioni non volute prima che la legge stessa venga emessa.
Inoltre, date le particolari caratteristiche del dominio legislativo, l'investigazione dei problemi di rappresentazione in tale campo riveste un interesse piu' generale /IJCA/. Il dominio legale è anche un campo privilegiato per la costruzione di sistemi basati su conoscenza in quanto la conoscenza è specificata in testi.
Questa particolarità da un lato costituisce un vantaggio rispetto ad altri campi applicativi e, dall'altro, è fonte di notevoli difficoltà, poichè devono essere affrontati i problemi relativi alla formalizzazione del linguaggio naturale e del commonsense reasoning.
Tra le caratteristiche della conoscenza contenuta in testi legislativi possono essere individuate :

- I termini utilizzati nei testi legislativi hanno sia un significato comune che un significato tecnico, in questo ultimo caso è necessario far riferimento a definizioni e specificazioni contenute in altri contesti normativi;
- la norma da applicare in una data circostanza puo' non essere esplicitamente definita ma spesso deve essere inferita da norme che fanno riferimento ad altre situazioni;
- la conoscenza è espressa a diversi livelli d'astrazione: oltre a regole che specificano definizioni e norme di comportamento esistono regole che specificano "principi generali" o che governano l'interpretazione di altre regole;

la conoscenza é soggetta a frequenti cambiamenti, dovuti all'emissione di leggi e circolari interpretative che integrano o modificano norme precedenti.

Per esplorare il potenziale della logica in quest'ambito abbiamo formalizzato in Prolog un frammento della legge che regolamenta il funzionamento dell'Università. La nostra ricerca è rivolta ai seguenti obiettivi:
- verificare in che misura le clausole di Horn sono un formalismo adeguato per la rappresentazione di norme legislative e per la costruzione di sistemi esperti di tipo legale;
- identificare meccanismi per rendere agevole la traduzione dei testi legislativi in programmi logici;
- costruire un setting sperimentale nel quale possano essere analizzati problemi di rappresentazione di conoscenza e tentativi di soluzione possono essere verificati.

Nel seguito é motivata la scelta di adottare un formalismo di tipo logico; successivamente viene descritto il frammento di legge che è stato preso in considerazione, la sua formalizzazione, e le linee metodologiche che sono state seguite; quindi sono illustrate alcune modalità di utilizzo della formalizzazione ed sono descritti i problemi di rappresentazione incontrati.

RAPPRESENTAZIONE LOGICA DELLE LEGGI

Diversi autori hanno proposto di adottare la logica simbolica come formalismo per la rappresentazione e l'analisi delle leggi /CIAM/,/NIBL/. Questo approccio appare promettente in quanto la logica fornisce un linguaggio formale che è particolarmente adatto per esprimere concetti di tipo definizionale che specificano relazioni e proprietà legali.
Inoltre la legislazione è spesso già espressa in una forma simile a quella logica e quindi una formalizzazione che mantenga la struttura del testo originale sarà più facile da verificare e da aggiornare in futuro al variare della legislazione.
Una formulazione logica di un insieme di norme ha una semantica dichiarativa comprensibile e manipolabile da un elaboratore. La disponibilità di interpreti efficienti per linguaggi basati su un sottoinsieme della logica del primo ordine (clausole di Horn) permette di utilizzare praticamente la rappresentazione della legge per costruire sistemi esperti di tipo legale. Tali sistemi possono applicare automaticamente la legge in specifiche circostanze e possono supportare la fase di stesura della legge permettendo di esplorare le conseguenze logiche della formalizzazione ed identificare potenziali implicazioni e contraddizioni.
Diverse esperienze di rappresentazione di leggi come programmi logici sono state effettuate all'Imperial College da /SE85/, /SSKK/ e all'Università di Waterloo da /HUST/.
Altre ricerche, basate su formalismi non logici, sono state condotte da McCarthy /McCa/, Waterman e Peterson /WAPE/ e Gardner /GARD/. Tra queste ultime l'esperienza più significativa è quella del progetto TAXMAN, che è rivolta a definire un modello concettuale mediante il quale rappresentare gli elementi dell'operare giuridico e sviluppare su di essi attività di ragionamento.
La nostra ricerca è più vicina al primo approccio, il nostro obbiettivo è individuare tecniche e metodologie per tradurre il contenuto dei testi legislativi in programmi logici.

LA LEGGE ANALIZZATA

La situazione che è stata presa in considerazione riguarda l'attribuzione di un insegnamento universitario vacante ad un docente.

L'attribuzione può essere effettuata secondo diverse modalità:

a) per "sostituzione"	i.e. il docente, al posto del corso di cui è titolare, tiene il corso vacante
b) per "secondo insegnamento"	i.e. il docente, oltre al proprio corso, è incaricato di tenere anche il corso vacante
c) per "supplenza"	i.e. il docente sostituisce il titolare del corso

L'utilizzo di una modalità è in relazione alle caratteristiche del corso, dell'Università di appartenenza e del docente.
I testi che contengono la normativa sono:
- la legge n.382 dell'11/7/1980 (art.9,25,29,100,114,116)
- la legge n.477 del 13/8/1984, (art.1,3) che modificano, rispettivamente, gli art. 9 e 114 della legge 382
- la circolare interpretativa della legge n.477 del 16/10/1984 riguardante il conferimento di supplenze

Si è scelto di rappresentare questo frammento di legge perchè esso presenta molte caratteristiche tipiche dei testi legislativi. La conoscenza è distribuita in diversi testi, contiene riferimenti a leggi precedenti ed è stata modificata più volte. Alcune norme sono definite esplicitamente mentre altre devono essere estratte da norme che regolano altre situazioni. La legge, oltre a contenere precise definizioni, esprime anche principi generali ed affermazioni di meta livello, e fa affidamento, in numerosi casi, a conoscenze implicite.

LA FORMALIZZAZIONE DELLA LEGGE

Anche in questo ambito, come in altri settori, un problema cruciale nella realizzazione del sistema riguarda l'acquisizione della conoscenza. Poichè la conoscenza non è contenuta in unico testo ma è distribuita in più leggi e, all'interno di ciascuna legge, in diversi articoli, la rappresentazione è stata effettuata secondo questi criteri metodologici:
a) dapprima è stato formalizzato separatamente ciascun articolo della legge 382,
b) le regole ottenute sono state coordinate, ed infine
c) la formalizzazione è stata modificata per tener conto della legge 477 e della circolare interpretativa.

La rappresentazione dei singoli articoli di legge è stata realizzata mediante l'analisi dei periodi che compongono il testo, ogni periodo del testo é stato tradotto in un frammento di programma Prolog. Ad esempio:

Art.114, comma 1, 1 parte

> Fino all'espletamento della prima tornata dei giudizi di idoneità per professore associato i posti di insegnamento rimasti vacanti per qualsiasi ragione, sempreché per l'insegnamento che si intende ricoprire per supplenza sia stato richiesto il posto di ruolo, e per i quali sia comprovata l'impossibilità di chiamata di professori di ruolo, possono essere conferiti per supplenza esclusivamente a professori ordinari e straordinari, a professori associati ovvero a professori incaricati stabilizzati; della stessa materia o di materia affine, appartenenti alla stessa facoltà;...

é rappresentato dalla clausola Prolog:

```
Insegnamento puo_essere_attribuito_in_supplenza_a Docente  :- (1)
        non_espletate_tornate_giudizi_di_idoneita,          (2)
        Insegnamento   e_vacante,                           (3)
        e_stato_richiesto_posto_di_ruolo_per  Insegnamento, (4)
        impossibilita_di_chiamata_per  Insegnamento,        (5)
        Docente  e_un_ord._o_staord._o_assoc._o_inc-stab,   (6)
        Insegnamento  appartiene_stessa_facolta_di Docente, (7)
        Insegnamento  e_stessa_materia_o_affine_a Docente.  (8)
```

Nella regola precedente il predicato (1) rappresenta l'azione regolamentata dalla legge, mentre i predicati (2)-(8) descrivono le condizioni che devono essere verificate affinché tale azione possa essere eseguita.
Si noti che ogni predicato corrisponde ad un frammento del testo analizzato, ad esempio: "e_stato_richiesto_posto_di_ruolo_per Insegnamento" rappresenta la parte di testo:
"... semprechè per l'insegnamento che si intende ricoprire per supplenza sia stato richiesto il posto di ruolo ..."

Ciascun predicato descrive una condizione elementare o complessa. Condizioni complesse sono costituite dalla congiunzione/disgiunzione di condizioni e sono specificate per mezzo di altre regole.
Ad esempio la condizione"Insegnamento e_stessa_materia_o_affine_a Docente"
é specificata dalla regola:

```
Insegnamento   e_stessa_materia_o_affine_a    Docente  :-
                    Insegnamento   e_stessa_materia   Docente ;
                    Insegnamento   affine_a       Docente.
```

Le affinità tra gli insegnamenti sono definite in un altra parte della legge in termini di appartenenza allo stesso gruppo concorsuale e, inserite nella rappresentazione come fatti, permettono di specificare ulteriormente le condizioni della regola precedente.

```
Insegnamento   affine_a   Docente  :-
        Insegnamento appartiene_raggruppamento_concorsuale X,
        Docente  insegna  Corso,
        Corso appartiene_raggruppamento_concorsuale X.
```

La traduzione della seconda parte dell'art. 114 è stata più problematica:

Art.114, comma 1, 2 parte

> ...Non possono comunque essere coperti per supplenza gli insegnamenti sdoppiati salvo che il numero degli esami sostenuti negli insegnamenti stessi nell'ultimo anno accademico sia superiore a 250 per ciascun corso attivato.

Questa norma descrive un eccezione alla regola precedente e dovrebbe essere rappresentata:
Insegnamento e_uno_sdoppiamento AND esami_insuff_in Insegnamento ⊃
⊃ NOT(Insegnamento puo_essere_attribuito_in_supplenza_a Docente)

Ma poichè in Prolog non sono ammesse regole con conclusioni negative, per rappresentare la norma esistono due possibilità:
1) trasformare la norma
2) esprimere la proibizione in un altro formalismo

Abbiamo sperimentato entrambe le soluzioni, descriviamo ora la prima e rimandiamo la presentazione della seconda alla sez.6.

E' stata definita una nuova regola che descrive l'eccezione:

```
Insegnamento non_attribuibile_in_supplenza_a Docente  :-
                    Insegnamento e_uno_sdoppiamento,
                    Numero esami_sostenuti_in Insegnamento,
                    Numero < 250.
```

e la sua conclusione negata è stata inserita nella parte condizionale della regola generale:

```
Insegnamento puo_essere_attribuito_in_supplenza_a Docente  :-
       non_espletate_tornate_giudizi_di_idoneita,
       Insegnamento   e_vacante,
       not(Insegnamento non_attribuibile_in_supplenza_a Docente),
                          ...
```

Nella regola precedente é utilizzato l'operatore "not", questo operatore non coincide con la negazione logica ma esprime "negazione come fallimento" /CLAR/ che e' consistente con la negazione logica sotto l'ipotesi della "closed world assumption".
Questo tipo di negazione e' particolarmente utile in ambito legale per la rappresentazione di situazioni eccezionali poiché spesso i testi legislativi definiscono una norma generale seguita da una lista di eccezioni. Se una situazione non soddisfa le condizioni che definiscono l'eccezione, é naturale considerarla come una situazione ordinaria.
"Negazione come fallimento" non é pero' sempre adeguata per rappresentare la proibizione, poiché essa non é applicabile qualora non si conoscano tutte le eccezioni rilevanti o sia necessario utilizzare due tipi diversi di negazione come é descritto da /SSKK/.

E' stata mostrata la formalizzazione dell'art. 114, le altre regole del sistema specificano le altre forme di assegnamento e definiscono concetti richiamati all'interno delle regole.
Poiche' nella legge non esiste un articolo che esplicitamente elenca i diversi modi con i quali il corso vacante puo' essere attribuito, é stato necessario definire una regola ad alto livello non corrispondente ad alcuna parte di testo.

```
Insegnamento puo_essere_attribuito_a Docente  :-
  Insegnamento puo_essere_attribuito_per_sostituzione_a Docente ;
  Insegnamento puo_essere_attribuito_per_2_insegn_a Docente ;
  Insegnamento puo_essere_attribuito_per_supplenza_a Docente.
```

La formalizzazione della legge si é conclusa con la rappresentazione del testo della circolare interpretativa che, come ci si aspettava, in molti casi, ha specificato per mezzo di altre regole concetti vaghi delle leggi precedenti che erano stati rappresentati come condizioni elementari.
La formalizzazione completa della legge, composta da 55 regole, che rappresentano circa sette pagine di testo, è descritta in /BABE/. L'implementazione è stata realizzata in Prolog II su un personal computer.

UTILIZZO DELLA FORMALIZZAZIONE

La rappresentazione della normativa puo' essere utilizzata per fornire risposte a quesiti che riguardino l'applicazione della legge in specifiche circostanze.
Ad esempio, si puo' verificare se un insegnamento puo' essere attribuito ad un docente secondo una particolare modalità o lasciando al sistema il compito di

determinare la modalità corretta e si possono richiedere gli insegnamenti ed i docenti che soddisfano i requisiti richiesti dalla legge.
La base di conoscenza come é stata ottenuta per mezzo della traduzione dei testi di legge non e' direttamente utilizzabile, per permettere la consultazione é necessario introdurre all'interno delle regole dei predicati di input/output e di controllo. Poiché l'introduzione di questi predicati distrugge la semantica dichiarativa della rappresentazione si è deciso di far carico all'interprete di provvedere a tali funzionalità.
E' stato sviluppato un interprete Prolog-like che incorpora una facility simile a "Query-the-user" implementata nel sistema APES /HASE/; mediante questa funzionalità i fatti sia positivi che negativi sono registrati nel database e, quando nel corso della dimostrazione, l'interprete non trova nel database un fatto né il suo contrario, interroga l'utente e trasforma la risposta ottenuta nel corrispondente fatto Prolog.

In relazione alla quantità di fatti inizialmente noti al sistema si hanno diversi tipi di utilizzo. In un primo tipo di utilizzo la base di conoscenza non contiene alcun fatto iniziale ed il sistema, per ciascuna condizione della regola alla quale la consultazione fa riferimento, richiede all'utente le caratteristiche della situazione da indagare, le risposte ottenute sono trasformate in fatti Prolog ed inseriti progressivamente nella base della conoscenza.
In un secondo tipo di utilizzo la base di conoscenza contiene una descrizione dettagliata degli insegnamenti e dei docenti della facoltà, queste informazioni saranno sfruttate per fornire una risposta ai quesiti richiesti senza ricorrere all'effettuazione di domande all'utente.
Un terzo genere di utilizzo prevede una base di conoscenza incompleta che non contiene tutte le informazioni richieste dalla legge. In questo caso, quando l'interprete nel corso della dimostrazione non troverà nella base di conoscenza il predicato corrispondente al tipo di requisito richiesto, la verifica sarà effettuata interagendo con l'utente con lo stesso meccanismo del caso 1.
Un esempio di consultazione é mostrato in fig.1., nella figura l'input dell'utente é sottolineato ed i commenti sono racchiusi tra "**".

PROBLEMI DI RAPPRESENTAZIONE

Nella traduzione delle norme sono stati incontrati diversi problemi di rappresentazione, tra di questi:
- l'impossibilità di rappresentare nelle regole Prolog conclusioni negative;
- la necessità di integrare la rappresentazione della conoscenza espressa in testi con definizioni, vincoli ... che non compaiono in alcun testo;
- la necessità di tradurre conoscenza di metalivello ad un livello oggetto o, mantenendo un doppio livello di rappresentazione, individuare meccanismi di interazione tra i livelli.

Per affrontare questi problemi é adottato un approccio sperimentale modificando la prima versione del sistema e costruendo diversi interpreti, ciascuno dei quali specializzato nella gestione di una particolare caratteristica. L'obbiettivo era di definire meccanismi di rappresentazione che permettessero di ridurre la quantità di interventi necessari per la costruzione della base di conoscenza.
Nella sez. 4 è stato mostrato come sia necessario trasformare una proibizione per l'impossibilità delle clausole di Horn di rappresentare conclusioni negative. Per evitare tali trasformazioni è stata inclusa nella base della conoscenza la proibizione di raggiungere una conclusione. In questa versione del sistema una conclusione di una regola può essere raggiunta se tutte le sue condizioni sono soddisfatte e non può essere provata alcuna conclusione contraria. Nel caso in cui nessuna conclusione negativa sia specificata il sistema si comporta come l'usuale interprete Prolog.
Questa soluzione ha il vantaggio che la formalizzazione risulta più simile


```
attribuzione insegnamenti vacanti.

Nome dell'insegnamento ?  analisi

Nome del docente ?  rossi

Quali modalità si intende utilizzare ?
        1- sostituzione
        2- secondo insegnamento
        3- supplenza
        4- da determinare

E' VERO CHE  non_espletate_tornate_giudizi_idoneita ?  si

E' VERO CHE  analisi e_uno_sdoppiamento ?  si

PER QUALE X: X esami_sostenuti_in analisi ?  300

        ...          ** verifica degli altri requisiti
                        dell'insegnamento   **


E' VERO CHE  rossi e_un_ordinario ?  no

E' VERO CHE  rossi e_un_associato ?  si


                  ** il sistema verifica autonomamente che:
                      - rossi appartiene alla stessa facoltà
                        di analisi
                      - rossi insegna una materia simile ad
                        analisi  **

CONCLUSIONE:  L'insegnamento analisi può essere attribuito in
              supplenza al docente rossi
```

fig.1

alla struttura del testo originario ma ulteriori approfondimenti sono necessari al fine di comprendere le sue limitazioni e le analogie con il concetto di "negazione come inconsistenza" /GASE/ e con altre forme di negazione avanzate recentemente /GARE/.
Un altro problema che è stato riscontrato é che le definizioni di alcune regole fanno affidamento ad assunzioni implicite non specificate in alcun testo. Ad esempio una norma dell'art. 9 stabilisce che un insegnamento può essere attribuito in supplenza ad un professore ordinario senza specificare che il corso deve essere vacante e, analogamente, nell'art. 114 non è specificato se è necessario il consenso del docente o se l'attribuzione deve essere effettuata su sua richiesta.
Questi esempi indicano che una base della conoscenza costituita unicamente dalla rappresentazione dei testi di legge non é sufficiente poichè conclusioni indesiderate possono essere successivamente derivate dalla formalizzazione.
Per permettere la rappresentazione di conoscenze implicite è stata fornita la possibilità di associare condizioni aggiuntive a ciascun predicato di una

regola. Le condizioni aggiuntive agiscono come vincoli che devono essere soddisfatti precedentemente o successivamente alla verifica del predicato a cui sono associati.

Le norme contenute nel testo spesso contengono dei requisiti riguardanti le azioni che devono essere eseguite per poter attribuire l'insegnamento. Questi requisiti sono di natura diversa dalle condizioni che specificano proprietà dell'insegnamento e del docente e, invece di rappresentarli assieme a queste ultime, essi sono stati isolati in modo da permettere consultazioni rivolte ad indagare entrambi gli aspetti, sia separatamente che in relazione tra di loro. In questo ultimo caso il sistema può fornire responsi di tipo "condizionale"

CONCLUSIONE: l'insegnamento analisi può essere attribuito al
docente rossi
MA le seguenti azioni devono essere eseguite:
- deve essere richiesto il posto di ruolo
- rossi deve presentare la domanda
- il Cosiglio di Facolta deve deliberare
l'assegnamento
...

Questi esempi hanno evidenziato che, per disegnare programmi logici che descrivono una normativa in modo preciso ed affidabile, la formalizzazione dei testi legislativi deve essere arricchita da una descrizione delle entità a cui le norme fanno riferimento. Inoltre strutture per la rappresentazione a diversi livelli di astrazione e meccanismi di interazione tra i livelli sono necessari. Attualmente stiamo analizzando questi problemi e stiamo sperimentando soluzioni rimanendo in un ambito logico.

CONCLUSIONI

Il dominio legale possiede particolari caratteristiche che lo differenziano da altri domini di conoscenza che sono stati analizzati e rappresentati, tra di queste di particolare importanza é il fatto che la conoscenza é espressa in testi scritti in linguaggio naturale.
Nell'articolo é stato presentato come delle norme legali possono essere formalizzate come programmi logici che, opportunamente arricchiti, costituiscono la base di conoscenza di un sistema esperto di tipo legale. In quest'ambito l'utilizzo di Prolog ha permesso uno stile di rappresentazione coinciso ed espressivo ed ha consentito uno sviluppo incrementale e per tentativi del sistema che si é dimostrato particolarmente adatto alla rappresentazione dei testi legislativi.

Ringraziamenti

Vorrei ringraziare il prof. G. Degli Antoni per i suggerimenti forniti nel corso della ricerca, A. Masia e O. Zancolò per l'aiuto prestato nella comprensione della legge e M. Stagnoli per la collaborazione allo sviluppo del sistema.

Referenze

/ASHL/ Ashley K.D. "Reasoning by analogy: a survey of selected AI research with implication for legal expert systems", Proc. 1° Conference on Law and Technology, University of Houston, 1985

/BABE/ Bandini S., Bertocchi R., "Rappresentazione di leggi in Prolog", in attesa di pubblicazione su 'Informatica e Diritto'

/CIAM/ Ciampi C. (ed.) "Artificial Intelligence and Legal Information systems", North Holland, 1982

/CLMc/ Clark K.L., Mc Cabe F. "Prolog for expert systems" in Machine Intelligence n.10, Ellis Horwood, 1982

/CLAR/ Clark K.L. "Negation as failure", in Logic and Data Bases,Gallaire H.(ed.), Plenum Press, New York, 1978.

/DEZO/ Degli Antoni G., Zonta B. "Analysis of law by means of Petri Nets: motivations and methodology", in Artificial Intelligence and Legal Information systems, Ciampi C. (ed.), North Holland, 1982

/GARD/ Gardner A. v.d.L. "The design of a Legal Analysis Program", Proc. AAAI-83, Washington, 1983

/GARE/ Gabbay D., Reyle U. "N-Prolog an extension of Prolog with hypothetical implications" , The Journal of logic programming, 1984, n. 3

/GASE/ Gabbay D., Sergot M. "Negation as inconsistency", Research Report Imperial College, 1984, London

/HASE/ Hammond P., Sergot M.J. "A Prolog shell for logic based expert systems", Proc. 3rd BCS Expert Systems Conference, Cambridge, British Computer Society, 1983

/HUST/ Hustler A. "Programming Law in Logic", Research Report CS-82-13, University of Waterloo, Canada , 1982

/IJCA/ Panel on Artificial Intelligence and Legal Reasoning, Proc. IJCAI-85, Los Angeles 1985

/McCA/ McCarthy L.T. "A computational theory of legal arguments", LRP TR 13, Laboratory for Computer Science Research, Rutgers University, 1982

/NIBL/ Niblett R. (ed.) "Computer Science and Law", Cambridge University Press, 1979

/PEOL/ Pereira L. M., Oliveira E. "Prolog for expert systems:a case study", Proc. IFAC, Leningrad, USSR 1983

/REIT/ Reiter R. "On closed world databases", in Logic and Data Bases,Gallaire H.(ed.), Plenum Press, New York, 1978.

/SE82/ Sergot M. J. "Prospects for representing the law as logic programs", in Logic Programming, Academic Press, London, 1982

/SE85/ Sergot M. J. "Representing legislation as logic programs", Research Report 1985, Imperial College, London

/SSKK/ Sergot M. J., Sadri F., Kowalski R.A., Kriwaczek F., Hammond P., Cory H.T., "The British nationality act as a logic program", Research Report 1985, Imperial College, London

/WAPE/ Waterman D.A., Peterson M.A., "Models of legal decision making", Report R-2717, Rand Corporation, Institute for Civil Justice, 1981

PRESENTAZIONE DEI SISTEMI DI INTELLIGENZA ARTIFICIALE TEKTRONIX COME SISTEMI ESPERTI DI DIAGNOSI GUASTI SU APPARECCHIATURE ELETTRONICHE.

ING.COSIMO PIERI E DOTT.MASSIMO BOMBANA
TEKTRONIX S.P.A. - MILANO

ABSTRACT

Come molte altre aziende e organismi di ricerca, la Tektronix ha studiato le potenzialita' dell'artificial intelligence per parecchi anni.
In questa presentazione viene descritto un esempio realizzato presso i Laboratori di Ricerca Tektronix di un prototipo di sistema esperto per aumentare la produttivita' dei settori dell'assistenza tecnica nella ricerca e diagnosi di guasti di complesse apparecchiature elettroniche(5).
A questo scopo viene fornita una descrizione dell'hardware utilizzato per la realizzazione del sistema esperto, una panoramica sui linguaggi di AI utilizzati sulle motivazioni che hanno portato alla loro scelta ed un sommario dello sviluppo del prototipo e della sua architettura.
Particolare rilievo viene dato all'interfaccia grafica, indispensabile per l'utilizzo pratico di sistemi analoghi.

IL SETTORE DI APPLICAZIONE: LA RICERCA E DIAGNOSI DEI GUASTI DI APPARECCHIATURE ELETTRONICHE.

In un'azienda come la Tektronix la ricerca e la diagnosi dei guasti sono un argomento di fondamentale importanza sia per l'uso interno sia per le applicazioni che questa tecnologia può trovare sul mercato.
In particolare e' abbastanza frequente che un buon tecnico esperto nella riparazione di complesse apparecchiature elettroniche possa avere l'opportunita' di passare, anche nella stessa azienda, a occuparsi di assistenza sistemistica o di assistenza alla vendita.
Quando questo accade le conoscenze acquisite da quello specifico tecnico sono il piu' delle volte perse.
E' quindi logico che uno dei primi tentativi di applicare le tecniche dell'intelligenza artificiale per la creazione di sistemi esperti sia stato proprio nel settore della diagnostica di apparecchiature elettroniche. Fino dalle fasi iniziali e' stata presente la convinzione che il ruolo di questi sistemi esperti non era quello di rimpiazzare i tecnici ma di aiutarli nello svolgere efficacemente il proprio lavoro.
Il sistema si comporta come una guida nei riguardi di tecnici "alle prime armi" aiutandoli a definire i problemi e le diagnosi cosi come potrebbe fare un tecnico di grande esperienza .
Nello stesso tempo anche tecnici esperti traggono indubbi vantaggi dalla possibilita' di mettere a disposizione di molti la loro conoscenza.
La tecnica dei sistemi esperti differisce dalla tradizionale programmazione per due aspetti:

1. La base di dati non consiste di dati numerici ma di un grande numero di regole.

2. Le decisioni vengono prese sulla base delle intuizioni e dell'esperienza degli esperti, piuttosto che sulla base di sole teorie formali.

IL PROTOTIPO DEL SISTEMA ESPERTO.

Quale primo prototipo e' stato scelto il sistema esperto per la ricerca e diagnosi dei guasti del generatore di funzioni Tektronix FG 502 (1, 2,5,11).
Questa prima realizzazione e' parte di un sistema piu' ampio in sviluppo presso la Tektronix, mirante a mettere a disposizione di tecnici e progettisti l'utilizzo di sistemi esperti per la riparazione di un determinato tipo di strumento.
In particolare e' stato giudicato di grande importanza nell'architettura del sistema esperto il tipo di interfacciamento tra il programma e l'utente.
Il dialogo viene semplificato mediante l'utilizzo di linguaggi naturali in studio presso gli stessi laboratori di ricerca Tektronix (7, 10).
L'interazione con l'utente viene anche facilitata mediante un'uso estensivo delle possibilita' grafiche offerte dal sistema 440X e dall'ambiente di programmazione.

L'ARCHITETTURA DEL SISTEMA

Questo sistema esperto e' stato realizzato mediante la workstation Tektronix per intelligenza artificiale 440X, sistema specificamente progettato per questo tipo di applicazioni.
Si tratta di una famiglia di prodotti, che consta dei tre modelli 4404, 4405 e 4406, che descriviamo brevemente di seguito. L'architetura di questi sistemi ha alcune caratteristiche comuni quali:

- Uso di una CPU 32 bit dotata di floating point hardware.
- Schermo grafico bit-mapped ad alte prestazioni specificamente progettato per i pop-up menus con 1024x1024 punti indirizzabili.
- Memoria centrale disponibile fino a 4,5 o 6 Mbytes a seconda dei modelli.
- Memoria di massa disponibile in linea fino a 270 Mbytes.
- Sistema operativo in grado di gestire, anche all'interno della stessa applicazione piu' linguaggi tra quelli disponibili per il sistema.
- Disponibilita', su tutti e tre i sistemi, di tutti i principali linguaggi per l'A.I. e cioe':
  - SMALLTALK - 80, come implementazione completa della versione originariamente sviluppata nei laboratori di ricerca della XEROX, particolarmente indicato per le notevoli capacita' grafiche e di object e incremental programming.
  - FRANZ LISP
  - COMMON LISP
  - MPROLOG
- Disponibilita' di compilatore C e, a breve, PASCAL.

In particolare:

- Il sistema piu' semplice e' il 4404, dotato di schermo 13", monocromatico e con 640x480 pixel, con CPU 68010 a 10 MHz, RAM da 1 a 4 Mbytes e sistema operativo con memoria virtuale di 8 Mbytes.
- Le altre due unita' utilizzano invece una CPU 68020 a 16 MHz con sistema operativo che estende in questo caso la memoria virtuale a 32 Mbytes. Lo schermo e' monocromatico di 13" (con 640x480 pixels per il 4405) e di 19" (con 1280x1024 pixels per il 4406). La memoria centrale va da 2 a 6 Mbytes per il modello 4406 e da 1 a 5 Mbytes per il 4405.

Vale la pena di sottolineare come, per ottimizzare le prestazioni del sistema esperto, sia stato possibile usare all'interno della stessa applicazione due dei vari linguaggi disponibili sulle workstations Tektronix 440X.
Infatti, mentre per le fasi di interazione grafica e l'implementazione dei linguaggi naturali e' stato usato il linguaggio Smalltalk-80*, il motore di inferenza del sistema e' stato implementato in Mprolog.**

* Smalltalk-80 e' un marchio registrato della Xerox Corporation.
** Mprolog e' un marchio registrato della Logicware.

LA REALIZZAZIONE DEL SISTEMA ESPERTO.

Si possono individuare quattro stadi nel processo che ha portato alla realizzazione di questo sistema esperto (5,9):

1. Studio del problema da affrontare, mediante approfonditi colloqui con tecnici del settore .
2. Formulazione di una strategia per automatizzare il processo. In questa fase sono state studiate le modalita' piu' efficaci di rappresentare e di utilizzare la conoscenza acquisita dai tecnici.
3. Sviluppo delle parti relative all'interazione tra l'utente e il sistema esperto.
4. Realizzazione del motore di inferenza, definendo e provando le regole, raffinandole via via che si inseriscono.

La definizione e la struttura della base della conoscenza e' fra le cose piu' critiche nel processo di sviluppo di un sistema esperto. Prolog e' un linguaggio che ben si adatta allo sviluppo di strategie di inferenza nel campo delle ricerche guasti in schede elettroniche, e per questo motivo e' stato scelto per l'applicazione in oggetto. Due motori diversi possono essere utilizzati: il primo, il piu' semplice, utilizza semplicemente la tecnica di ricerca top-down, mentre il secondo, piu' avanzato, attua una strategia che permette di utilizzare delle ipotesi relative a direzioni di esplorazione privilegiate nelle ricerche del guasto. In (5) viene fornito un esempio semplificato di codice prolog per la realizzazione di questi due diversi tipi di motori di inferenza.
In quest'ultimo caso e' necessaria una ulteriore conoscenza sotto forma di indicazioni euristiche, che permettano al motore di inferenza di funzionare correttamente ed in modo efficente.

Il sistema e' in grado di supportare l'acquisizione di nuove regole che vengono aggiunte alla base di conoscenza gia' presente. Questa caratteristica permette in particolare di modificare senza troppa fatica il sistema, per adattarlo a nuove esigenze.
L'interfaccia INGLISH (IO) permette di tradurre le espressioni del linguaggio corrente in clausole prolog, le quali vengono inserite nel motore di inferenza che opera in backgroung.
Le informazioni vengono trasmesse dall'ambiente Smalltalk all'ambiente Prolog, e viceversa, mediante l'uso di classi che permettono di aprire e chiudere canali di collegamento a livello di sistema operativo. Un passo successivo in questa direzione sara' compiuto creando vere e proprie finestre Prolog in ambiente Smalltalk. Il sistema, ancora allo stato di prototipo, si e' gia' rivelato estremamente utile soprattutto per la semplicita' d'uso (dovuta all'interfaccia in linguaggio naturale) e per l'efficenza (dovuta alla scelta di prolog per la realizzazione del motore di inferenza).

LA RAPPRESENTAZIONE GRAFICA.

La notevole capacita' di rapprensentazione grafica dei sistemi (si può arrivare a 1280x1024 pixels visibili) e le efficaci possibilita' con varie finestre anche sovrapponibili per menu' dinamici, hanno permesso di realizzare un sistema caratterizzato da un utilizzo molto agile e diretto.
In particolare possono essere contemporaneamente visualizzate le rappresentazioni degli schemi funzionali dei circuiti elettronici, le piastre dei circuiti stampati, come pure le regole del funzionamento del circuito e i comandi operativi del sistema.
L'utilizzo del mouse e di simboli grafici permette inoltre di simulare efficacemente l'interazione con il circuito visualizzando le sonde e gli altri strumenti utilizzati.
Risulta particolarmente interessante seguire una sessione di ricerca guasti, come può essre facilmente presentata utilizzando uno dei tre sistemi della serie 440X e di cui alleghiamo un esempio in figura 1.

CONCLUSIONE

Analisi di mercato (9) fanno prevedere una forte espansione nel prossimo quinquennio (1985-1990) delle richieste da parte del mondo produttivo di sistemi esperti implementati su workstations caratterizzate come piccoli sistemi stand-alone single-user e dal costo limitato.
Tektronix ha dimostrato con la serie 440X di aver interpretato per tempo questa tendenza, e di avere fornito un hardware adatto ad applicazioni di questo tipo.
Un sistema esperto come quello descritto e' stato sviluppato in un tempo relativamente breve per assistere i tecnici di manutenzione nella ricerca guasti.
Ulteriori applicazioni in questo settore sono previste in un prossimo futuro per uso sia interno che esterno all'azienda.

BIBLIOGRAFIA

1) Alexander, J.H.& Freiling, M.J. (1985) Smalltalk-80 aids troubleshooting system development. System and Software 4,4.

2) Alexander, J.H., Freiling, M.J., Messick. S.L., & Rehfuss, S. (1985) Efficient expert system development through domain specific tools. Fifth International Workshop on Expert System and their Applications.

3) Buchanan. B.G., Barstow, D., Bechtal. R., Bennett, J., Clancey, W., Kulikowski, C., Mitchell, T., & Waterman, D.A. (1983) Constructing an Expert System. In F.Hayes-Roth, D.A.Waterman, & D.B. Lenat (Eds.), Building Expert System. Reading, MA: Addison-Wesley.

4) Burton, R.R. (1976) Semantic Grammar : An engineering technique for constructing natural language understanding systems. Tech. Rep. 3453 Bolt. Beranek, and Newman, Cambridge, MA,

5) Freiling M., Alexander J., Messik S., Rehfuss S., Shulman S. (1985) Starting a Knowledge Engineering Project: a step-by step approach. AI Magazine Fall, 1985.

6) Freiling. M.J. & Alexander. J.H. (1984) Diagrams and grammars: tools for the mass production of expert systems. First Conference on Artificial Intelligence Applications. IEEE Computer Society.

7) Freiling. M.J., Alexander. J.H., Feucht. D., & Stubbs. D. (1984) GLIB- a language for describing the behavior of electronic devices. Applied Research Tech. Rep. CR-84-12 Tektronix.Inc., Beaverton, OR.

8) Goldberg, A. & Robson. D. (1983) Smalltalk 80: the language and its implementation. Reading, MA: Addison-Wesley.

9) Harman P., King D. (1985), Expert System: AI in business. J.Wiley & Sons, Inc.

10) Phillips, B. & Nicholl. S. (1984) INGLISH : A natural language interface. Applied Research Tech. Rep. CR-84-27. Tektronix ,Inc., Beaverton, OR.

11) Phillips, B., Messick, S.L., Freiling, M.J., & Alexander. J.H. (1985) INKA: The Inglish knowledge acquisition interface for electronic instrument troubleshooting systems. Applied Research Tech. Rep. CR-85-04, Tektronix, Inc, Beaverton, OR.

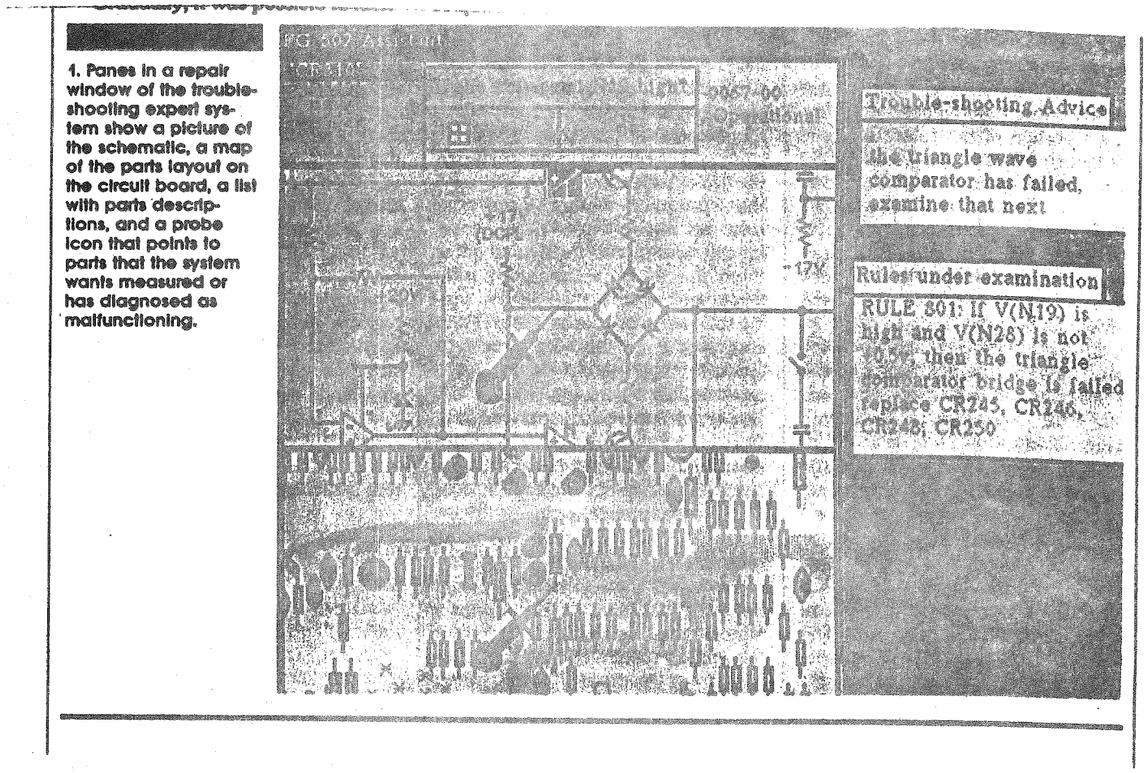


FIGURA 1

PROTOTIPAZIONE VELOCE IN PROLOG: UN SISTEMA ESPERTO NELLA ACQUISIZIONE DELLA CONOSCENZA

M. MISSIKOFF , G. SISSA

IASI - CNR

VIALE MANZONI 30 ROMA

SOMMARIO

Il presente articolo illustra una esperienza di prototipo veloce, sviluppata nell'ambito del progetto MOSAICO.

Il sistema MOSAICO, attualmente in corso di realizzazione presso lo IASI-CNR di Roma, si pone l'obiettivo di affiancare l'esperto di Analisi dei Requisiti e di agevolarne il lavoro, usando tecniche di Intelligenza Artificiale.

Mediante una interfaccia semplice ed agevole da usare l'utente puo' introdurre le informazioni sulla realta' sotto osservazione ed il sistema provvede ad organizzarle in modo da costruire progressivamente una rete di concetti: la Base di Conoscenza di MOSAICO.

In questo articolo il fuoco e' sull' organizzazione della Base di Conoscenza ed in particolare sulle strutture dati progettate per rappresentare la rete di concetti e sulle tecniche utilizzate per implementare dette strutture dati utilizzando il linguaggio PROLOG.

INTRODUZIONE

Il progetto MOSAICO nasce con l'obiettivo di realizzare un sistema per automatizzare la fase preliminare della progettazione di un Sistema Informativo (SI): la raccolta dei requisiti di utente e la stesura delle specifiche preliminari del SI (User and Application Requirement (UAR) Specification).

Oggi l'analisi e' portata avanti da specialisti che utilizzano soprattutto la loro esperienza e qualche metodo empirico e parziale.

Le specifiche, che rappresentano il prodotto dell'analisi di cui sopra, sono rappresentate da uno o piu' rapporti tecnici essenzialmente redatti in linguaggio naturale, con l' aiuto di diagrammi a chiarimento

del testo.

Questo modo di affrontare la prima fase di un progetto di SI si e' ampiamente dimostrata carente, sia per l' inesistenza di un metodo sufficientemente rigoroso da garantire un'analisi esauriente del problema, sia per l'uso del linguaggio naturale nella stesura delle specifiche, che e' un mezzo espressivo poco adatto a rappresentare conoscenza in modo rigoroso. Infine, data la scarsa strutturazione del contenuto di un rapporto tecnico, la ricerca di una singola informazione appare spesso lunga e tediosa.

L'Intelligenza Artificiale (IA) da tempo vede tra i suoi obiettivi principali la rappresentazione della conoscenza, soprattutto in ambiti complessi o parzialmente definiti.

In un' ottica di IA la fase di raccolta dei requisiti di utente si colloca nel filone dell' Acquisizione della Conoscenza e la stesura delle specifiche in quello della costruzione di Basi di Conoscenza (BC).

E' in quest' ottica che il progetto MOSAICO prende l'avvio con una strategia essenzialmente "bottom up", ossia avendo come primo obiettivo la definizione di un modello di conoscenza, cioe' un insieme di formalismi atti a rappresentare i diversi aspetti e le connessioni esistenti tra le entita' di una realta' complessa, del tipo di quella generalmente affrontata nel progettare un SI.

Il progetto MOSAICO e' centrato sulla generazione di una Base di Conoscenza che contenga una descrizione del dominio di applicazione (detta Target Knowledge Base: TKB). La Base di Conoscenza deve essere sufficientemente espressiva da raccogliere descrizioni complesse e semanticamente ricche della realta'.

Poiche' MOSAICO opera essenzialmente su conoscenza, la sua expertise viene trattata come metaconoscenza e costituisce la Meta Knowledge Base (MKB). La MKB e' organizzata secondo un modello, detto Meta Knowldge Model (MKM), che consente di rappresentare la realta' secondo quattro categorie concettuali: entita', azioni, eventi e processi [MISS85].

Mediante le categorie citate e' possibile rappresentare diversi aspetti della realta'. Aspetti fattuali, modellando concetti e collegamenti fra di essi; aspetti dinamici, rappresentando la conoscenza causale e restrittiva; infine aspetti estensionali, consentendo di usare l' esemplificazione per la verifica e, in certi casi, la generazione di concetti.

Il presente articolo affronta i temi piu' rilevanti dell' esperienza di prototipazione rapida della sezione di MOSAICO dedicata alla gestione delle entita'.

Il prossimo paragrafo e' dedicato alla descrizione del formalismo scelto per la rappresentazione della conoscenza.

Il paragrafo successivo descrive le diverse strutture dati sulle quali, nella fase di progetto di MOSAICO, si e' pensato di centrare l'implementazione del KBMS (Knowledge Base Management System); dopo una fase di analisi e sperimentazione si e' optato per la struttura ad albero

AND-OR, che appare la piu' consona all'ambiente PROLOG in cui MOSAICO viene sviluppato.

Segue poi il paragrafo che illustra in maggiore dettaglio la struttura dati di base, concentrando l' attenzione in particolare sulla parte di Base di Conoscenza relativa alle entita'.

L'ultimo paragrafo descrive brevemente l'ambiente di sviluppo, il POPLOG, utilizzato nella fase attuale del nostro lavoro.

LA RAPPRESENTAZIONE DELLA CONOSCENZA

Il formalismo di rappresentazione della conoscenza utilizzato fa riferimento ai frames.

I frames, introdotti inizialmente da Minsky [MINS75], costituiscono una struttura informativa concepita per rappresentare situazioni ed entita' stereotipate.

In generale un frame puo' essere pensato come una rete gerarchica in cui i nodi rappresentano concetti, proprieta', informazioni e gli archi le relazioni che tra questi intercorrono. Nel procedere dall'alto verso il basso i nodi rappresentano via via concetti sempre meno generali, fino ad arrivare ai nodi terminali a cui vengono associati i valori di istanze reali (dati fattuali).

In MOSAICO la Base di Conoscenza viene organizzata come una rete di frames. Il singolo frame contiene e organizza le informazioni di stretta pertinenza del concetto che implementa; ma un concetto e' definito utilizzando altri concetti, ed esso, a sua volta, contribuisce a definire altri concetti. In quest'ottica nel frame vi trovano posto sia informazioni strettamente locali, cioe' che contribuiscono direttamente alla definizione del concetto, sia informazioni di riferimento ad altri concetti; queste ultime consentono la creazione della rete concettuale.

Qui di seguito illustriamo, come esempio, l'organizzazione di un frame atto a rappresentare concetti del tipo "entita'".

FRAME : ENTITA'

SLOT NOME: ...
SLOT PROPRIETA':...
SLOT AZIONI:...
SLOT REGOLE:...
SLOT GERARCHIE:...
SLOT ECCEZIONI:...
SLOT DEFAULT:...
SLOT ESEMPI:...

Per quanto riguarda un concetto di tipo entita' dovranno essere specificate le informazioni relative al nome, alle sue proprieta', alle azioni in cui e' coinvolto, alle regole che si applicano a differenti elementi di conoscenza del concetto rappresentato, ai legami gerarchici (es. generalizzazioni), alle descrizioni di eventuali eccezioni, a possibili valori di default e range di valori ammessi, ad esempi significativi di istanze dell' entita' in oggetto.

Per ogni entita' della Base di Conoscenza vi sara' un frame contenete le informazioni necessarie a definire l' entita', opportunamente organizzate mediante gli slot appena illustrati.

I concetti relativi alle altre categorie (azioni, eventi, processi) sono rappresentati mediante strutture analoghe.

La BC di Mosaico e' organizzata come una rete semantica, i cui nodi sono rappresentati da frames del tipo descritto.

Oltre ai collegamenti nello slot gerarchie con concetti piu' generali o piu' specifici, gerarchie "is-a", vi sono anche collegamenti con concetti diversi, ma allo stesso livelo di astrazione, tramite un esplicito riferimento dallo slot opportuno. Nel nostro modello ogni concetto deve essere collegato ad almeno un altro concetto. Questo significa che la rete semantica complessiva deve formare un grafo connesso.

Oltre a quanto gia' descritto, l'utente ha la possibilita' di associare alle proprieta' di una data entita' ulteriori informazioni, riguardanti le caratteristiche della proprieta' stessa.

Tali caratteristiche consentono di indicare se una proprieta':

- fa riferimento ad un'altra entita'
- rappresenta una proprieta' strutturata, ossia articolabile in modo ulteriore indicando le informazoni piu' elementari che la definiscono (concetto di molecola). La scelta di introdurre la molecola come qualcosa di concettualmente intermedio fra l' elemento atomico e la vera e propria entita' se da un lato rende piu' complessa la struttura dall'altro consente di modellare in modo piu' preciso ed espressivo la conoscenza.
- potra' assumere piu' di un valore, quando istanziata.
- e' marcata come proprieta' definita in modo tentativo, da rivedere in momenti successivi, durante il raffinamento del progetto della Base di Conoscenza.

Si e' stabilito che le proprieta' possano godere di piu' caratteristiche.

Data una stuttura informativa del tipo descritto sono stati pensati vari metodi implementativi possibili.

L' ORGANIZZAZIONE DELLA BASE DI CONOSCENZA

Data una struttura informativa ricca e complessa come quella descritta nel paragrafo precedente, si e' posto il problema di scegliere il tipo di struttura dati piu' adatta ad implementarla.

Questa scelta e' stata effettuata tenendo conto che l'attivita' si colloca nell' ambito di una esperienza di prototipazione rapida mediante PROLOG e pertanto si svolge nell' ottica di privilegiare la velocita' di sviluppo, garantendo gli aspetti funzionali, a scapito dell' efficienza complessiva (tempi di risposta, occupazione di memeoria, ecc) [RAMA84].

Nell'implementazione delle strutture informative si sono presentate tre alternative: il record, la lista, l'albero AND-OR.

Record

E' sicuramente la struttura piu' utilizzata nel software applicativo; essa appare particolarmente adatta a rappresentare informazioni con una struttura relativamente semplice ed altamente ripetitiva, si pensi ad un archivio del personale di un' azienda, o alla gestione magazzino.

Il frame entita' appare molto articolato. Esso contente informazioni di tipo molto diverso, e soprattutto di dimensioni difficilmente stimabili a priori: infatti possono esistere concetti semplici e concetti complessi, similmente la struttura deve poter adattarsi alle diverse situazioni. L' organizzazione a record e' apparsa troppo rigida da questo punto di vista; inoltre con i record l' elaborazione avviene sul principio della ciclicita', ovvero della scansione ciclica dei dati e del test di condizioni (spesso su flag) che scatenano le azioni opportune. In sostanza le azioni non sono attivate dalla conformazione della struttura (troppo povera appunto), ma piuttosto dal contenuto di determinati campi. Questo vuol dire che le informazioni contenute nella struttura possono avere sia il compito di rappresentare conoscenze sul mondo esterno, sia di controllare e condizionare l'elaborazione.

A questo tipo di soluzione, detta "content directed processing", per sistemi complessi quale il nostro, e' preferibile una soluzione del tipo "structure directed processing" dove le elaborazioni sono associate alle porzioni delle strutture informative visitate, la cui semantica ne risulta pertanto chiaramente determinata.

Lista

Questa struttura appare molto piu' flessibile della precedente e pertanto piu' adatta ad essere utilizzata per implementare un frame. Se adottata in modo flessibile e completo, cioe' consentendo l'uso di liste di liste in modo ricorsivo, si ha una struttura estremamente flessibile e potente in grado di modellare con estrema aderenza diversi concetti, con diversi gradi di complessita'. In quest'ottica la struttura a liste appare in un certo senso basilare. Il problema a questo punto e' quello di implementare il software di gestione di frames a partire da una struttura

in cui le primitive disponibili (del tipo: top, append, ecc.) sono troppo elementari. Inoltre la potenza di determinati meccanismi del PROLOG, in una soluzione a liste viene scarsamente utilizzata.

Albero AND- OR

Una struttra ad albero AND-OR si colloca da un livello intermedio rispetto alle due precedenti. Essa appare sicuramente piu' flessibile della struttura a record, dal momento che consente la realizzazione di sottostrutture alternative e ripetitive in modo estremamante naturale.

Rispetto alla organizzazione a liste, l'albero AND-OR appare piu' strutturato e, soprattutto, la manipolazione avviene utilizzando massicciamente le carateristiche del PROLOG in cui le sottostrutture in AND sono riportate come termini a destra di una clausola composta, mentre strutture in alternativa danno luogo a diverse clausole composte aventi la parte sinistra eguale.

Nel paragrafo successivo illustriamo il frame entita' organizzato secondo i legami propri di un albero AND-OR.

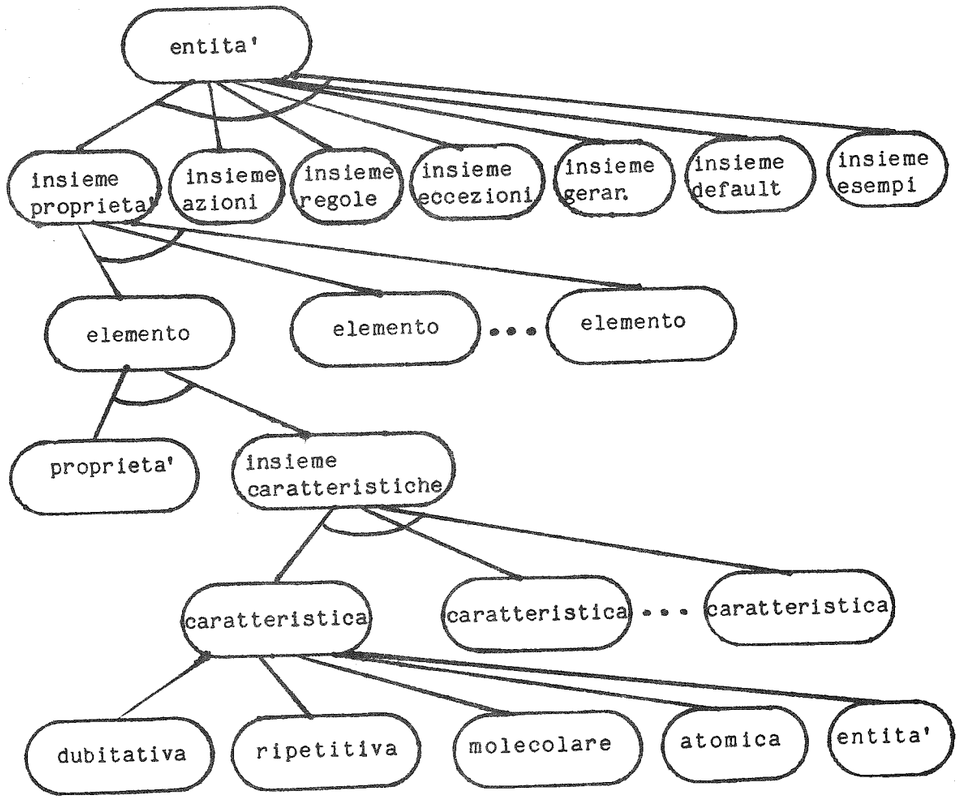
L'IMPLEMENTAZIONE DEL FRAME "ENTITA'"

Fatta la scelta implementativa di utilizzare gli alberi AND-OR, il frame relativo alle entita' risulta strutturato come indicato nella figura seguente. Tale albero visualizza in modo sintetico quanto detto nel paragrafo precedente.
La scrittura della corrispondente struttura PROLOG e' immediata.

E' stato cosi' possibile, a partire da tali scelte implementative, realizzare in tempi assai rapidi un sistema di lettura, scrittura e modifica, nonche' selezione di frames in base a criteri specificati dall' utente.

E' stata realizzata una raccolta di primitive compatte e del tutto generali, che saranno riutilizzate nella stesura delle altre sezioni del sistema. Questo, a nostro avviso, costituisce una indicazione di lavoro metodologica molto importante lavorando in un' ottica di prototipazione veloce.

Il codice prodotto e' scritto interamemte in PROLOG per consentirne la portabilita'.



L'AMBIENTE DI PROTOTIPAZIONE RAPIDA: IL SISTEMA POPLOG

Il sistema POPLOG, realizzato alla Sussex University e disponibile su DEC VAX, sotto VMS (o UNIX), costituisce un ambiente integrato di linguaggi: algoritmico (POP11), funzionale (LISP) e logico (PROLOG).

Il POPLOG consente la gestione integrata dei diversi linguaggi mediante video editor interattivo: il VED; questo consente inoltre la gestione di un multiwindow virtuale (massimo due window fisiche).

Il PROLOG di POPLOG consente la compilazione incrementale producendo un codice intermedio portabile (POPLOG virtual machine).

In questo ambienti sono disponibili inoltre: un ricco ambiente di aiuto (400 teach files e 720 help files) e varie librerie di routines, tra cui routines per l'interfaccia in linguaggio (quasi) naturale, nuclei di sistemi esperti, gestione ATN, gestione liste ed altro.

Si prevede che in fase di stesura definitiva del sistema MOSAICO si

fara' un uso piu' esteso del POPLOG, sia per funzioni di back-end, nella gestione della BC su memoria di massa, che di front-end, ad esempio sfruttando le caratteristiche del video editor interattivo per una interfaccia amichevole.

BIBLIOGRAFIA

[CLOC81] W. F. CLOCKSIN, C. S. MELLISH: "Programming in Prolog", Springer-Verlag, 1981.

[MELL84] C. MELLISH, S. HARDY: "Integrating PROLOG in the POPLOG environment", in "Implementation of PROLOG", pp.145,172 ellis Horwood Series AI, 1984.

[MINS75] M. MINSKY: "A Framework for Representing Knowledge", in "The Psychology Computer Vision", P. Winston(Ed.), Mc Graw Hill,1975.

[MISS85] M. MISSIKOFF, G. WIEDERHOLD : "The Design of MOSAICO: an Expert System for User and Application Requirement Analysis", Rapp. Tec. IASI, DIC 85.

[RAMA84] C.V. RAMAMOORTHY, ATUL PRAKASH, WEI-TEK TSAI, YATUKA USUDA : "Software Engineering: problems and perspectives", in "Computer", October 1984.

[SLOM83] A. SLOMAN, S. HARDY, J. GIBSON : "A multilanguage programdevelopmentenvironment", in "Information Technology: Reasearchand Development", 2, pp. 109, 122, 1983.

UN MODULO DI SPIEGAZIONE PER SISTEMI ESPERTI

Franco Cecchini

SIPE OPTIMATION S.p.A.
Laboratorio di Ricerca e Sviluppo

ABSTRACT

Uno dei campi di ricerca nell'Intelligenza Artificiale è costituito dai sistemi esperti i quali, basandosi su di una base di conoscenza e su capacità deduttive, risolvono problemi fornendo consulenze su particolari settori. Una delle caratteristiche essenziali di tali sistemi è data, dal momento che essi si rivolgono nella maggior parte dei casi ed utenti non esperti, dalla capacità di fornire una spiegazione delle azioni intraprese: ciò aiuta l'utente ad una maggiore comprensione del dominio del problema portandolo così a convincersi che le conclusioni raggiunte dal sistema sono basate su di un ragionamento accettabile. Il modulo presentato in questo articolo è appunto un sistema di spiegazione disegnato per spiegare il comportamento di sistemi esperti scritti in Prolog.

1. Introduzione

Le ricerche nel campo dell'Intelligenza Artificiale hanno come obiettivo lo studio e la costruzione di sistemi che agiscano con un certo grado di "intelligenza", tali cioè che il loro comportamento in determinate circostanze sia, almeno per certi aspetti, indistinguibile da quello che un essere umano avrebbe tenuto nelle stesse circostanze. Uno di questi comportamenti è la capacità di usare una base di conoscenza, insieme a capacità deduttive, per risolvere problemi. I sistemi che sono dotati di

queste capacità sono detti "sistemi esperti" in quanto riescono a dare consulenze su particolari settori, come esperti umani. Affinchè tale consulenza sia completa e di facile comprensione anche per gli utenti non esperti del dominio (ai quali i sistemi esperti si rivolgono nella maggior parte dei casi), questi sistemi dovrebbero avere la capacità di fornire una spiegazione per giustificare in modo comprensibile il loro comportamento sia per convincere l'utente dell'esattezza del ragionamento seguito durante il processo deduttivo sia per aiutarlo a risolvere il suo problema [Hayes]. Moduli di spiegazione sono stati costruiti per molti sistemi quali, ad esempio, EMYCIN, KAS ed EXPERT, dove vengono forniti sistemi interattivi per l'analisi delle conclusioni raggiunte. In questo articolo viene illustrato un sistema generale disegnato per spiegare il comportamento di sistemi esperti scritti in Prolog. Tale modulo di spiegazione compie un ragionamento a livello meta sullo spazio di ricerca visitato dal sistema esperto in modo da tenere traccia delle deduzioni effettuate per poi trattare tali deduzioni in modo da fornire una spiegazione sul perchè un particolare goal è stato soddisfatto o meno. Questa spiegazione viene generata sulla base di una descrizione della base di conoscenza su cui si basa il sistema esperto che viene fornita dallo specialista applicativo che codifica tale conoscenza. Tale spiegazione avviene interattivamente con l'utente, fornendo cioè una prima risposta generale al suo goal e poi chiedendo cosa va spiegato in modo più dettagliato. All'utente viene inoltre concessa la possibilità di modificare la granulità della spiegazione specificando cosa deve essere spiegato e come.

2. Requisiti e scelte di progettazione

I requisiti più importanti per un modulo di spiegazione sono:

1. la sua indipendenza dal sistema esperto specifico e la sua compatibilità con esso;
2. una comprensibilità definita in termini del dominio di applicazione piuttosto che del linguaggio di calcolo;
3. la capacità di ragionare sul comportamento del sistema esperto realizzando differenti strategie di spiegazione.

Per soddisfare il primo requisito si è scelto di operare al meta-livello [Aiello], cioè di progettare il modulo di spiegazione come un sistema esperto capace di ragionare sul comportamento di un sistema esperto. Il meta-livello, infatti, permette di considerare le azioni intraprese dal sistema esperto, cioè consente dal modulo di spiegazione di avere conoscenza dei metodi di inferenza usati, facilitando così la ricostruzione dei passi deduttivi con cui il sistema esperto ha interpretato una richiesta

e la presentazione "ragionata" delle deduzione all'utente. Quindi tale approccio offre il chiaro vantaggio di rendere il modulo di spiegazione indipendente dal particolare sistema esperto.
Per integrare conoscenza (quella del dominio su cui si basa il sistema esperto) e meta-conoscenza (quella del modulo di spiegazione) si possono seguire due strade:

1. codificare la meta-conoscenza direttamente nella base di conoscenza del sistema esperto;
2. separare i due livelli di conoscenza in moduli separati.

La scelta effettuata è stata di seguire la prima strada per il vantaggio che essa offre nel controllo dei predicati built-in Prolog quali, per esempio, il cut. Il meta-livello viene incorporato nella base di conoscenza del sistema esperto tramite una fase di precompilazione di tale conoscenza in cui le regole ed i fatti del dominio vengono riscritti inserendo ogni subgoal come argomento di una clausola la cui forma è: **meta_explain(subgoal).**
Il meta livello avrà il compito di memorizzare lo spazio di ricerca (un albero and-or) visitato dal sistema esperto durante le sue deduzioni tramite delle asserzioni del tipo:

answer (node_num,level_num,node_type,goal)

dove node_num è il numero del nodo visitato nell'albero, level_num è il numero del livello (distinguendo così quali sono i nodi allo stesso livello), node_type descrive l'evento associato al nodo e goal è l'instanziazione del goal Prolog di quel nodo. L'evento associato al nodo (node_type) può assumere quattro diversi valori (CALL, EXIT, REDO, FAIL) in accordo al modo in cui il Prolog tenta di soddisfare un goal [Klock, Kowa]. L'ultimo argomento (goal) è costituito, come detto, dall'instanziazione del goal Prolog del nodo considerato, in quanto la spiegazione deve mostrare i nodi istanziati in accordo alla soluzione raggiunta dal sistema esperto. Le deduzioni effettuate vengono asserite e non memorizzate in un parametro del meta-livello in quanto nel secondo caso si perderebbe traccia degli eventuali fallimenti dei quali, invece, è importante avere la spiegazione: l'utente è infatti spesso più interessato a conoscere le motivazione che hanno condotto ad un fallimento della sua richiesta piuttosto che ad avere una spiegazione del perchè una sua richiesta è stata soddisfatta.
La comprensibilità definita in termini del dominio di applicazione piuttosto che del linguaggio di calcolo è necessaria in quanto la presentazione della semplice "traccia" delle deduzioni effettuate dal sistema esperto o delle regole e fatti usati durante tali deduzioni richiederebbe, da parte dell'utente, una conoscenza del linguaggio di calcolo e di come è stata codificata la base di conoscenza [Davis]. Per raggiungere tale scopo, lo specialista applicativo fornisce una descrizione della base di conoscenza su

cui poi si baseranno le spiegazioni.
Quindi vi sono due fasi di uso del modulo di spiegazione: la prima di istanziazione da parte dello specialista applicativo che codifica la conoscenza necessaria al modulo per fornire la spiegazione, e la seconda di uso da parte dell'utente (o dello stesso specialista per controllare la correttezza della base di conoscenza da lui codificata) a cui viene presentata una spiegazione basata sulle deduzioni effettuate dal sistema esperto tradotte in termini delle descrizioni fornite in precedenza.

3. Descrizione della base di conoscenza

Come detto la prima fase di uso del modulo di spiegazione consiste in una istanziazione, da parte dello specialista applicativo, della descrizione della base di conoscenza. Questa fase avviene interattivamente: il modulo presenta allo specialista le clausole presenti nella base di conoscenza chiedendogli delle descrizioni. Le clausole non vengono presentate in modo top-down, cioè così come sono scritte nel file, ma seguendo la struttura logica della base di conoscenza. Infatti, seguendo l'ordine piatto in cui sono state codificate le clausole, si perde quel filo logico seguito invece durante la fase di codifica. Seguire la struttura logica della base di conoscenza vuol dire visitare staticamente una foresta di alberi, in cui le radici corrispondono alle varie richieste che possono essere fatte al sistema esperto. Affinchè il modulo abbia conoscenza delle radici dei vari alberi, occorre che esse siano indicate dallo specialista durante la fase di codifica del sistema esperto.
Per ogni nodo dell'albero lo specialista deve prima di tutto specificare il suo grado di visibilità; egli può in questo modo indicare al modulo che una clausola è completamente visibile (cioè all'utente va spiegata sia la chiamata della clausola che tutti i suo sottoscopi), semivisibile (con ciò si indica che di quella clausola devono essere spiegate solo le chiamate) o invisibile (la clausola non dovrà mai comparire durante la spiegazione). In questo modo lo specialista può nascondere all'utente quella parte di base di conoscenza che per lui non avrebbe significato (ad esempio di una regola che esegue dei calcoli numerici l'utente potrebbe essere interessato a vedere solo il risultato, e in questo caso lo specialista la indicherebbe come semivisibile). Dopo tale indicazione, per le clausole aperte all'Utente, va indicata una descrizione del funtore rappresentante la parte verbale della frase che dovrà spiegare quella clausola.
Gli argomenti della testa della clausola possono poi essere tipicizzati in quattro modi deversi: nodescription (cioè andrà mostrata la sola istanziazione di quell'argomento senza la sua descrizione), novalue (cioè non dovrà essere mostrato nè l'argomento nè la descrizione), value (verrà mostrato sia

l'argomento che la sua descrizione) o list (indicando così che l'argomento è una lista).

```
Functor --  lezione_accettabile
Visibility (v,s,i)? v
Verb phrase? può fare lezione

Type of the argoment n.1 is nodescription, novalue, value
or list? value
The argument n.1 is the subject (y/n)? y
Description of the argument n.1? Il professore

Type of the argument n. 2 is nodescription, novalue, value
or list? value
The argument n. 2 is the subject (y/n)? n
Description of the argument n. 2? alla classe

Type of the argument n. 3 is nodescription, novalue, value
or list? value
The argument n. 3 is the subject (y/n)? n
Description of the argument n. 3? nell'aula

Type of the argument n. 4 is nodescription, novalue, value
or list? value
The argument n. 4 is the subject (y/n)? n
Description of the argument n. 4? alle

Type of the argument n. 5 is nodescription, novalue, value
or list? value
The argument n. 5 is the subject (y/n)? n
Description of the argument n. 5? il giorno
```

Fig. 1

Nel primo e nel terzo caso viene poi richiesto di indicare se quell'argomento costituisce il soggetto della frase (non necessariamente unico) o un oggetto, e poi viene richiesta la sua descrizione. Nel quarto caso occorre tipicizzare ulteriormente l'argomento indicando se la lista è costituita da argomenti distinti o tutti dello stesso genere (per esempio una lista di nomi di professori), e in accordo a tale indicazione vanno poi fornite le tipicizzazioni degli elementi della lista.
Nella figura 1 è mostrato un esempio di tale fase di istanziazione per una regola Prolog la cui testa è:


```
lezione_accettabile(Professore,Classe,Aula,Ora,Giorno)
```

in questo esempio un goal del tipo:

```
lezione_accettabile("Rossi","I-A","S-1",9,"lunedi'")
```

verrebbe spiegato, nel caso di successo (EXIT), con la frase:
il professore Rossi può fare lezione alla classe I-A nell'aula S-1 alle 9 il giorno lunedi
o, nel caso di un fallimento (FAIL), con la frase:
il professore Rossi non può fare lezione alla classe I-A nell'aula S-1 alle 9 il giorno lunedi

Questa descrizione semplifica inoltre la fase di precompilazione, in quanto il modulo compila, della base di conoscenza, solo ciò che è stato "aperto" all'utente (cioè solo le clausole indicate come visibili o semivisibili) ed in accordo a come ciò è stato indicato.

4. Spiegazione delle deduzioni

In questa seconda fase di uso del modulo viene presentata all'utente, se da lui richiesto, una spiegazione delle deduzioni effettuate dal sistema esperto, in accordo alla descrizione della base di conoscenza fornita dallo specialista. Tale spiegazione si svolge interattivamente con l'utente: gli viene cioè fornita una prima risposta al suo goal, e poi gli viene richiesto di indicare, se lo desidera, cosa va spiegato in modo più dettagliato.
La spiegazione viene cioè ottenuta estraendo dall'insieme di assenzioni fatta dal meta-livello quelle corrispondenti ai nodi del goal dell'utente ed ai suoi figli diretti (tecnicamente è come spiegare un certo risultato, positivo o noegativo, è stato ottenuto tramite l'applicazione di una certa clausola), e ragionando poi su tali nodi.
Tale ragionamento viene eseguito tramite due diverse strategie di spiegazione a seconda dell'esito positivo o negativo di ciò che deve essere espiegato. Senza scendere troppo nei dettagli si può dire che nel caso di un successo vengono spiegati, tra i subgoals corrispondenti a nodi figli diretti, solo quelli che hanno condotto ad un successo; se si è verificato invece un fallimento, vengono spiegati sia i subgoals che hanno condotto ad un successo sia quelli che hanno invece portato ad un insuccesso. Queste due strategie trattano in modo differente i nodi etichettati come REDO, in quanto la prima deve spiegare solo l'ultimo (cioè perchè tutto quello che precede tale nodo costituisce un falso fallimento), mentre la seconda deve spiegarli tutti.

L'utente può chiedere due tipi di spiegazione:

- WHY per avere una spiegazione globale sia dei successi che dei fallimenti;
- WHYNOT per ottenere la spiegazione solo di ciò che ha condotto ad un fallimento.

Durante la fase di spiegazione l'utente può, se lo desidera, modificarne la granularità: egli può aggiungere un ulteriore livello di visibilità a ciò che gli era stato completamente "aperto" dallo specialista (cioè alle clausole etichettate come "visibili"). Quindi egli può modificare il livello di visibilità di una clausola (a cui accede tramite una delle frasi che gli vengono mostrate) in modo da modellare, secondo le sue esigenze particolari, la spiegazione. Nella figura 2 è mostrato un esempio di spiegazione.

```
lezione_accettabile("Rossi","I-A","S-1",9,lunedi").
YES
WHY.

1 Il professore Rossi può fare lezione alla classe I-A nell'aula
  S-1 alle 9 il giorno lunedi perchè

2 la classe I-A è libera alle 9 il giorno lunedi
3 l'aula S-1 è libera alle 9 il giorno lunedi
4 la classe I-A ha una numerosità uguale a 30
5 l'aula S-1 ha una capienza uguale a 35
6 35 non è minore di 30

WHY (2).

1 la classe I-A è libera alle 9 il giorno lunedi perchè
2 nessun professore fa lezione alla classe I-A alle 9 il giorno
  lunedi
```

Fig. 2

---

5. Conclusioni

Questo modulo di spiegazione, per le sue caratteristiche, si pone quindi come uno strumento rivolto all'utente di un sistema esperto il quale, anche se non esperto nè del dominio del linguagggio di programmazione può, grazie ad esso, venire aiutato

nel comprendere il dominio e nella costruzione incrementale della base di conoscenza. Vari sono gli sviluppi previsti per tale modulo; uno, ad esempio, è quello di rendere il modulo di spiegazione capace di spiegare non solo il perchè di un certo risultato, ma anche l'eventuale conoscenza non esplicita del dominio: ad esempio potrebbe essere utile avere la spiegazione anche del perchè una certa clausola è presente nel sistema esperto. Un altro aspetto allo studio è quello di rendere il modulo di spiegazione un vero sistema esperto sul dominio del sistema esperto su cui esso si basa, sgravando in una certa misura il compito dello specialista: cioè il modulo dovrebbe avere una maggiore conoscenza sintattica e semantica del dominio facilitando così l'inserimento della descrizione della base di conoscenza.

6. Bibliografia

(Aiello) L. Aiello, G. Levi, "The use of metaknowledge in AI systems", ECAI 84: Advanced in Artificial Intelligence, ED. T. O'Shea, 1984, pp. 705-717

(Clock) W. F. Clocksin, C.S. Mellish, "Programming in Prolog", Springer Verlag, New York, 1981

(Davis) R. Davis, D.B. Lenat, "Knowledge-Based Systems in Artificial Intelligence", McGraw-Hill International Book Company, 1982

(Hayes) F.Hayes-Roth, D.A.Waterman, D.B. Lenat, "Building Expert Systems", Addison-Wesley, 1983

(Kowa) R. Kowalski, "Logic for Problem Solving", North Holland, Amsterdam, 1979

PROGRAMMAZIONE LOGICA NEL PROGETTO EVA

F. EUSCONI, M. ONETO, G. VIANO

Servizio Ricerca Centralizzata
Elettronica San Giorgio - ELSAG S.p.A.

ABSTRACT

Il progetto EVA, Esperto di Visione Artificiale, si propone di interpretare scene complesse costituite da pezzi meccanici 'quasi piatti', comunque disposti su un piano. Deve percio' trattare situazioni non ben definite a priori e quindi affrontare un tipo di problemi non risolvibili con algoritmi tradizionali.

Nel presente articolo viene descritto il progetto, esponendo i problemi specifici affrontati e le soluzioni adottate. In particolare si da' rilievo all'utilizzo della Programmazione Logica, evidenziando alcune caratteristiche che hanno permesso di definire soluzioni efficaci sia nell'impostazione metodologica che concettuale.

1 INTRODUZIONE

Nell'ambito delle ricerche sulla Visione e Intelligenza Artificiale, si e' sviluppato in ELSAG il progetto EVA (Esperto di Visione Artificiale). Questo progetto e' finalizzato alla realizzazione di un sistema di visione applicabile in ambito industriale per localizzare e identificare pezzi meccanici.

Lo sviluppo di un tale sistema richiede l'utilizzo di metodologie diverse, dall'elaborazione di tipo numerico dei dati piu' vicini all'immagine, al trattamento di informazioni in forma simbolica per i livelli di maggiore astrazione.

Per affrontare quest'ultimo tipo di elaborazione si e' ritenuta particolarmente adeguata la Programmazione Logica. L'articolo si propone di illustrare il progetto dando maggior rilevanza alle parti di esso sviluppate con tale metodologia.

2 APPROCCIO AL PROBLEMA

2.1 DESCRIZIONE DEL PROGETTO EVA

Il progetto EVA, pur avendo principalmente un obiettivo di ricerca, si propone la realizzazione di un sistema di visione collocabile in un contesto applicativo ben definito.
Le sue specifiche prevedono di riconoscere pezzi 'quasi bidimensionali' (con un'altezza inferiore alle altre dimensioni e sufficiente solo a proiettare piccole ombre) e di identificare la presenza di eventuali oggetti tridimensionali. Non vengono posti vincoli significativi ne' sull'ambiente (illuminazione, punto di vista, etc.), ne' sulla disposizione degli oggetti (fattore di scala, sovrapposizione, etc.).
Compito di EVA e' quindi di interpretare scene complesse che possono comprendere piu' oggetti orientati in modo casuale, eventualmente visibili in modo parziale a causa di fenomeni di sovrapposizione e/o parzialmente deformati da piccole ombre o riflessi.
Il sistema fa uso di modelli degli oggetti e dell'ambiente, che possono cambiare con l'applicazione specifica; contiene una descrizione dei modelli che viene prodotta automaticamente da una procedura di acquisizione off-line.

2.2 PROBLEMATICHE CARATTERISTICHE DEL PROGETTO EVA

I problemi affrontati da un sistema di visione di un certo grado di complessita', come e' questo, sono numerosi e di diversa natura [NEVA 78, TSOT 82]. Di seguito ne viene fornita una panoramica schematica ed il piu' possibile completa.

Il primo punto che si e' dovuto affrontare e' stato come descrivere gli oggetti. Le caratteristiche che il formalismo di descrizione deve avere sono:

- generalita': la descrizione non deve essere specifica di un singolo dominio di applicazione, ma adattabile a diverse esigenze;

- sinteticita': poiche' si ha a che fare con una enorme quantita' di dati, occorre trovare un modo per riassumerli ottimizzando l'occupazione di memoria con la minima perdita di informazione;

- stabilita': la descrizione deve essere il piu' possibile insensibile alle alterazioni introdotte dagli strumenti utilizzati per l'acquisizione (sia "hardware" che "software") ed anche alle eventuali deformazioni causate da fenomeni fisici quali la distorsione prospettica, la presenza di ombre ,riflessi, sovrapposizioni, etc.;

- chiarezza : la descrizione deve contenere in modo esplicito e facilmente accessibile le caratteristiche distintive dell'oggetto;

- localita': la descrizione deve consentire di rappresentare porzioni dell'oggetto in modo da permettere il riconoscimento di oggetti visti anche solo parzialmente;

- dinamicita': la descrizione deve poter rappresentare conoscenze dinamiche in modo da potersi adattare a diversi domini di applicazione ed alle diverse fasi di analisi e riconoscimento.

Altre problematiche sono invece tipiche del processo di riconoscimento. Esse riguardano proprio le diverse azioni che deve fare un riconoscitore e cioe':

- ritrovare nella descrizione del dominio dati estratti dall'immagine (problema del ritrovamento di informazioni);

- confrontare i dati estratti dall'immagine con quelli relativi al dominio; questo presuppone che le descrizioni siano omogenee;

- muoversi all'interno di un grande spazio di ricerca. Questo problema diventa critico con l'aumentare del numero di pezzi da riconoscere, con le ambiguita' strutturali degli oggetti stessi (oggetti che sono simili, che hanno parti di contorno uguali, proporzionali, etc.) ed eventualmente a causa di errori generati dai processi di livello basso (vedi par. 3.2.1) che accrescono il numero di alternative;

- mettere insieme diverse fonti di conoscenza che vanno da quelle proprie del dominio di applicazione a conoscenze a priori su diversi fenomeni fisici quali occlusioni, proiezione di ombre, etc.;

- usare meccanismi di ragionamento per l'interpretazione dei dati; questo e' necessario per poter assegnare un significato ai dati esaminati quando le informazioni estratte dall'immagine sono incomplete, ambigue e/o rumorose e quindi non coincidenti direttamente con i modelli memorizzati.

2.3 SOLUZIONI ADOTTATE

Dall'analisi dei problemi esposti sono derivate in modo naturale alcune scelte riguardanti il modello di rappresentazione delle conoscenze, le strategie da adottare, i formalismi da utilizzare.

Per la rappresentazione delle conoscenze, si e' individuato come elemento caratterizzante l'utilizzo di entita' simboliche nelle descrizioni. In particolare gli oggetti sono descritti in modo gerarchico [SHAP 80, FU 74] come insiemi di forme chiuse e queste come sequenze di primitive geometriche, caratterizzate da una parte simbolica ed una numerica (vedi par. 3.1). Questo consente di rispondere in particolare ai requisiti di sinteticita', chiarezza e localita'.

Riguardo alla strategia da adottare, una volta definite le diverse azioni che concorrono al riconoscimento, si sono associati ad ognuna di esse processi "specialisti" che lavorano in modo autonomo; essi sono coordinati da un supervisore e dialogano tramite una base dati comune.

Si e' poi pensato di codificare i criteri decisionali (sia quelli del supervisore che quelli dei vari "specialisti") mediante un formalismo a regole. Questo formalismo e' stato ritenuto il piu' adatto a rappresentare la conoscenza decisionale, cioe' quella che tende a sfoltire lo spazio di ricerca, ad adattare la fase di ragionamento alle diverse situazioni possibili ed a codificare le euristiche utilizzate.

Le interazioni tra i diversi tipi di conoscenza, sia dichiarativa che procedurale, si sono risolte con la scelta di un formalismo di rappresentazione omogeneo.

2.4 SCELTA DEL PROLOG

La scelta di un linguaggio di Programmazione Logica, e quindi del Prolog, e' parsa la piu' naturale visto il tipo di approccio con cui si e' affrontato il problema [KOWA 79, CLAR 82]. Il Prolog possiede molti dei requisiti necessari: esso infatti consente di trattare dati simbolici, di gestire dati dinamici (liste, clausole) e di creare programmi in modo modulare, permettendo la coesistenza di processi autonomi all'interno di un unico sistema; ha una sintassi particolarmente semplice che rende possibile un formalismo omogeneo tra diversi tipi di dati e tra dati e programmi e consente di codificare e processare in modo immediato le regole; possiede inoltre un database con i relativi strumenti di accesso ad esso, che si e' rivelato particolarmente utile nella nostra applicazione [CLOC 81, KLUZ 85].

Altri linguaggi avrebbero potuto ugualmente fornire una risposta alle nostre esigenze, ma in modo meno diretto e naturale del Prolog.
Ad esempio il Lisp possiede tutti i requisiti citati fatta eccezione per la presenza di un motore inferenziale per processare le regole, cosa che il Prolog possiede gia' e la cui implementazione sarebbe risultata onerosa oltre che estranea al progetto.
Linguaggi piu' tradizionali, come il Pascal, avrebbero richiesto anche la costruzione di un database e di primitive di manipolazione di dati simbolici e dinamici; inoltre nessuno di questi presenta omogeneita' di formalismo.

Altre caratteristiche proprie del Prolog si sono rivelate di estrema utilita' durante la realizzazione del lavoro. Le elenchiamo di seguito schematicamente:

- la possibilita' di aver a disposizione la ricerca automatica nella base dati mediante i meccanismi di "matching" e di unificazione;

- la possibilita' di utilizzare il meccanismo di controllo come criterio di attivazione di regole e di ricerca nella base dati;

- il "rapid prototyping" dei programmi dovuto alla semplicita' della sintassi, alla modificabilita' dei predicati, alla possibilita' di creare programmi in modo incrementale e di testarli via via;

- l'immediatezza dell'approccio dichiarativo che ha consentito in varie situazioni di effettuare una traduzione "testuale" di un algoritmo, cosa impossibile in un qualsiasi altro linguaggio.

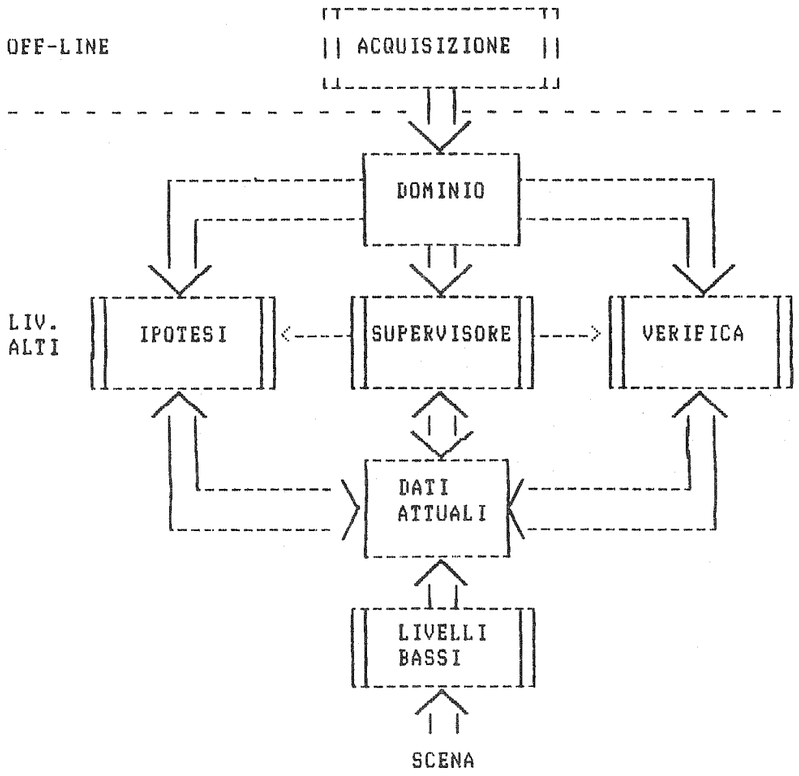
3 STRUTTURA DEL SISTEMA

Il Sistema e' costituito da varie parti, secondo lo schema di fig 1.

3.1 BASE DATI

La base dati contiene tutti i dati a disposizione del sistema.

Distinguiamo logicamente due diversi tipi di dati, il primo relativo all'ambiente di applicazione (dati a priori o dominio), il secondo alle conoscenze attuali sulla scena in esame (dati attuali).



- Fig. 1 - EVA: Schema Generale del Sistema -

Nel dominio sono descritti gli oggetti in modo gerarchico, strutturati cioe` in diversi livelli. Un oggetto e` descritto come insiemi di forme (contorni chiusi) e relazioni tra esse; queste a loro volta sono descritte come sequenze di primitive geometriche e loro relazioni. Le primitive che sono alla base di questa gerarchia rappresentano porzioni limitate del contorno, rispettando la caratteristica di localita` richiesta dalla descrizione; sono caratterizzate da una parte simbolica (per es. 'retta', 'angolo destro ottuso', 'curva sinistra', etc.) quantificata da un insieme di parametri (per es. lunghezza, apertura angolare, curvatura, etc.).

I dati attuali comprendono le conoscenze correnti sulla scena in esame. Essi si evolvono man mano che l'analisi procede e rispecchiano lo stato della conoscenza che si ha della scena. Si passa cosi` gradatamente da una descrizione della scena in termini di primitive geometriche a quella piu` sintetica e significativa contenente oggetti e relazioni spaziali tra essi. Poiche` l'analisi della scena non procede in modo sequenziale, nella base dati si possono avere contemporaneamente i diversi livelli di descrizione (dalla descrizione di porzioni di scena in termini di primitive geometriche all' individuazione di ombre, riflessi, overlap, etc.); questo e` possibile grazie all'omogeneita` del formalismo di rappresentazione.

L'implementazione di queste rappresentazioni in Prolog utilizza predicati opportunamente strutturati.
Ad esempio i modelli degli oggetti sono descritti dal predicato 'oggetto', che ha per argomenti l'identificatore, le liste delle forme componenti, delle caratteristiche globali e delle relazioni tra componenti. Le forme sono descritte in modo analogo, con un predicato 'forma', avendo pero` come componenti le primitive. Le primitive sono rappresentate esplicitamente nei dati attuali anch'esse come predicati contenenti per argomenti i loro parametri simbolici e numerici. Ad esempio il predicato:

```
primitiva(p1, cos, [3.37E+1,1.79E+1,1.51E+2,3.2E+2,146,303]).
```

descrive una primitiva con identificatore 'p1', parte simbolica 'cos' indicante una Curva che sottende un angolo Ottuso ed ha orientazione Sinistra, lista dei parametri indicanti lunghezza dell'arco equivalente, raggio di curvatura, coordinate del centro e coordinate del punto iniziale nel sistema di riferimanto dell'immagine.

Per memorizzare e gestire la nostra base dati abbiamo utilizzato il database del Prolog e la sua caratteristica di essere un 'data-base management system'. Nel database del Prolog, insieme alle regole ed ai diversi programmi, coesistono i dati del dominio ed i dati relativi alla conoscenza che si ha sulla scena.

3.2 PROCEDURE DI ANALISI DELLA SCENA

Le procedure di analisi della scena si possono distinguere in procedure di livello basso, di livello alto e supervisione.

3.2.1 LIVELLO BASSO

I processi di livello basso operano sulla rappresentazione digitalizzata dell'immagine acquisita dalla telecamera; attraverso successive elaborazioni esse costruiscono una descrizione della scena in termini di contorni chiusi caratterizzati dai loro angoli, rette e curve, ottenendo una sintesi delle informazioni rilevanti presenti [DUDA 73, PAVL 77]. Tale descrizione costituisce il primo livello memorizzato nella base dati corrente, che viene aggiornata e accresciuta dai processi di livello alto.
I processi di livello basso richiedono elaborazioni di tipo prevalentemente numerico [HOGG 84], che si prestano ad essere trattate con metodologie tradizionali (Von Newmann languages); questa e` percio` l'unica parte del sistema interamente non implementata in Prolog [CAVI 86].

3.2.2 LIVELLO ALTO

Il compito di comprensione della scena e` svolto da processi di livello alto che lavorano in modo concorrente coordinati da un supervisore.
Ognuno di questi processi esegue delle specifiche operazioni sui dati dell'immagine, secondo un paradigma di ipotesi e verifica.

In generale i processi di generazione di ipotesi si occupano di suggerire la presenza nella scena di entita' (forme, oggetti) o fenomeni (sovrapposizioni, ombre, riflessi, istabilita' nella descrizione), usando i dati sintattico-geometrici a disposizione (processi data-driven). Non tutte le ipotesi teoricamente possibili vengono considerate, ma soltanto quelle maggiormente evidenti.
Per la parte attualmente implementata, riguardante le ipotesi di forma, l'algoritmo si puo' ricondurre ad un confronto combinatorio di liste e loro sottoinsiemi, e da' come risultato l'insieme delle sottoliste comuni a contorni dell'immagine e modelli, selezionando quelle piu' promettenti secondo un criterio di affidabilita'. Attraverso queste sottoliste vengono associati all'immagine dei modelli di forma con la loro posizione, orientazione, fattore di scala. Le forme istanziate possono essere viste anche solo parzialmente.

I processi di verifica si occupano di controllare la plausibilita' delle ipotesi emesse dai moduli precedenti utilizzando i modelli (processi model-driven).
Per la parte riguardante il riconoscimento di forme, l'algoritmo si occupa di completare i modelli visti solo parzialmente recuperando, se possibile, eventuali porzioni di contorno non ancora associate al modello in esame.

3.2.3 SUPERVISORE

Il supervisore gestisce l'attivazione e l'interazione dei processi di livello alto, e la loro comunicazione con la base dati corrente. Esso implementa i criteri decisionali ed alcune delle euristiche presenti in modo da restringere lo spazio di ricerca sulle possibili ipotesi di interpretazione, realizzando cosi' il miglior compromesso tra risorse impiegate (tempo, memoria etc.) e qualita' della risposta fornita.

Per questo modulo e' particolarmente adeguato il formalismo a regole, che consente di codificare il comportamento del sistema nelle diverse situazioni senza predefinire la sequenza in cui le operazioni dovranno essere effettuate. Inoltre e' possibile ottimizzare le prestazioni fornite modificando in modo semplice l'insieme di regole e verificandole immediatamente, sfruttando le caratteristiche offerte dal linguaggio Prolog.

3.3 ACQUISIZIONE DEL DOMINIO

La procedura di acquisizione e' utilizzata off-line ogni volta che si vuole affrontare una nuova applicazione cambiando dominio. Essa produce automaticamente la descrizione dei modelli utilizzando processi di livello basso analoghi a quelli dell'analisi, cui segue l'apprendimento delle caratteristiche significative degli oggetti.

Al fine di facilitare la trasportabilita' e migliorare le prestazioni del sistema, sara' possibile estendere questa componente, e ottenere cosi' capacita' di autoapprendimento e autoadattamento off-line sul dominio applicativo.

4 CONCLUSIONI

La scelta del Prolog per le fasi di studio e per la realizzazione di una prima versione sperimentale del progetto EVA si e' rivelata conforme alle migliori aspettative, consentendo anche una rapida implementazione ed una veloce sperimentazione delle varie parti.
Accanto a questi vantaggi, sono stati pero' rilevati alcuni aspetti negativi in gran parte dovuti all'assenza di un prodotto consolidato. In particolare si e' riscontrato che, nella sua veste attuale, il Prolog puo' essere utilizzato in pratica solo per lavori di ricerca metodologica, in quanto le prestazioni in termini di tempi di esecuzione non sono buone.

Si e' inoltre verificato che i programmi Prolog diventano poco leggibili quando raggiungono un certo grado di complessita', rendendo arduo il lavoro di revisione. L'implementazione del Prolog a nostra disposizione elimina in parte questo inconveniente consentendo una programmazione a moduli; questo pero' in un certo senso falsa la filosofia del linguaggio stesso e lo rende meno pulito in quanto implica l'uso di istruzioni diverse dai predicati (ad esempio dichiarazioni di interfacciamento tra i moduli e dichiarazioni riguardanti lo scopo delle variabili).
Un ulteriore risvolto negativo e' derivato dalla semplicita' della sintassi che rende pesante e spesso illeggibile, senza una chiave di lettura appropriata, la descrizione di oggetti complessi.

In generale si puo' tuttavia affermare che l'esperienza fatta finora sulla Programmazione Logica, applicata ad un tema di ricerca nella visione, e' risultata positiva e fornisce buoni presupposti per gli sviluppi previsti.

5 BIBLIOGRAFIA

[CAVI 86] Caviglione, M. - Pardo, A. 'Il Modulo per la rappresentazione simbolico-parametrica dei contorni', Rel. Int. SRC/147, 1986.
[CLAR 82] Clark,K.L. - Tarnlund, S.A. 'logic Programming', Academic Press, New York, 1982.
[CLOC 81] Clocksin, W.F. - Mellish, C.S. 'Programming in Prolog', Springer Verlag, Berlin, 1981.
[DUDA 73] Duda, R.O. - Hart, P.C. 'Pattern Classification and Scene Analysis', John Wiley Inc, U.S.A., 1973.
[FU 74] Fu, K.S. 'Syntactic Method in Pattern Recognition', Academic Press, 1974.
[HOGG 84] Hogger, C.J. 'Introduction to Logic Programming', Academic Press, London, 1984.
[KLUZ 85] Kluzniak, F. - Szpakowicz, S. 'Prolog for Programmers', Academic Press, London, 1985.
[KOWA 79] Kowalski, R. 'Logic for Problem Solving', Elsevier Science Publishing Co., New York 1979.
[NEVA 78] Nevatia, R., 'Characterization and Requirements of Computer Vision Systems', in Hanson-Risemann 'Computer Vision Systems', New York, 1978.
[PAVL 77] Pavlidis, T. 'Structural Pattern Recognition', Springer Verlag, New York, 1977.
[SHAP 80] Shapiro, L.G. 'A structural model of shape', IEEE transaction on PAMI, Vol 2 n 2, March, 1980.
[TSOT 82] Tsotsos, J.K. 'Knowledge of Visual System: Content form and use', proc of IAPR, Munich, 1982.

SISTEMA DI RICONOSCIMENTO DI IMMAGINI NMR

G.Armano , S.Dellepiane , S.B.Serpico , G.Vernazza
Dip.to di Ingegneria Biofisica ed Elettronica
Via all'Opera Pia 11a, 16145 Genova

ABSTRACT

L'approccio integrato tra pattern recognition e intelligenza artificiale, sta rivelandosi particolarmente promettente ed innovativo in varie applicazioni.
Nell'ambito di tale metodologia, e' stato sviluppato un sistema che riconosce organi in immagini NMR del cranio.
Il riconoscitore e' basato su regole di produzione,struttura eterarchica e data base di tipo blackboard.
Il linguaggio adottato per il riconoscimento degli organi (alto livello) e' il PROLOG, mentre il linguaggio impiegato per l'estrazione di parametri dalle immagini e' il FORTRAN 77 (basso livello).
In questo lavoro si pone in evidenza l'utilita' del linguaggio PROLOG nelle applicazioni di riconoscimento.

1. INTRODUZIONE

Il riconoscimento di immagini costituisce una tra le piu' interessanti applicazioni dell' intelligenza artificiale e del pattern recognition. Argomenti principali della ricerca su tale problema sono lo sviluppo di architetture hardware che permettano tempi di calcolo accettabili, di strumenti software atti a rappresentare la conoscenza ed i suoi meccanismi, e la scelta delle tecniche di analisi del contenuto di informazione dell' immagine.
Il presente lavoro si incentra sugli ultimi due aspetti, affidandosi, al momento attuale, ad elaboratori ad architetture convenzionali, peraltro inadeguate per applicazioni su larga scala.
I linguaggi piu' diffusi nel settore dell' intelligenza artificiale sono indubbiamente il LISP ed il PROLOG. Il primo e' basato su un approccio simbolico funzionale, mentre il secondo, come ben noto, e' piu' propriamente un linguaggio di programmazione logica. Tra gli scopi del lavoro, si vuole verificare la validita' della scelta del linguaggio PROLOG per problemi di riconoscimento, e confrontare i risultati ottenuti con tale linguaggio con quelli ottenuti utilizzando il LISP.
Le principali tecniche di analisi di immagini orientate al pattern recognition sono il metodo statistico e quello strutturale (sintattico e relazionale).
Nel primo caso agli oggetti da riconoscere viene associato un vettore di misure (feature) in base al quale gli oggetti sono classificati in classi predefinite dall' operatore (metodo supervisionato) o in classi naturali (metodo non supervisionato). Gli oggetti da classificare in tal caso sono le regioni in cui l' immagine puo' essere suddivisa.
L' analisi strutturale permette di sfruttare le informazioni relazionali sugli oggetti da riconoscere. In particolare vengono considerate le relazioni tra le dimensioni, la posizione, la composizione. In tal modo si riconosce un oggetto in un contesto, e non in base alle sole proprieta' intrinseche. Strumenti atti a rappresentare tali informazioni sono i grafi relazionali, gli alberi sintattici, le reti semantiche [1].
Si ritiene particolarmente interessante l' approccio che integra le tecniche

del pattern recognition con quelle dell'intelligenza artificiale, al fine di introdurre un'opportuna rappresentazione della conoscenza, un meccanismo di inferenza e strategie di controllo [2]. Come applicazione di tale metodica, e' stato sviluppato un sistema che opera su immagini NMR [3] di sezioni del cranio.

2. SVILUPPO DEL SISTEMA

Dopo la prima fase di estrazione di regioni elementari (segmentazione) ed associazione di un vettore di misure a ciascuna di esse (fase non considerata in questo contesto) viene applicato, al fine del riconoscimento, un insieme di clausole PROLOG che realizzano l' equivalente di un sistema a regole di produzione dotato di meccanismo di focalizzazione, data base di tipo blackboard e struttura di controllo eterarchica.
In base ai valori assunti dai vettori di misure, alcune regioni vengono etichettate come regioni caratteristiche (regioni allungate, regioni curve, regioni piccole, regioni estese, regioni chiare, regioni scure, etc..). Particolari combinazioni di caratteristiche per una stessa regione (ad es. regione estesa, curva e chiara) candidano la regione ad essere riconosciuta come appartenente ad un organo.
L' effettivo riconoscimento viene effettuato dal corrispondente Sottosistema di Riconoscimento di Organo (SRO) in base ad ulteriori misure e relazioni con le altre regioni riconosciute. Il riconoscimento delle prime regioni fornisce informazioni utili per il riconoscimento delle restanti (informazioni strutturali). Gli SRO sono costituiti da regole di produzione che inglobano la conoscenza a priori sulla sezione da riconoscere. L'interprete decide volta per volta quale SRO deve essere attivato in base alla situazione attuale delle candidature e del riconoscimento, registrata nella blackboard. La blackboard costituisce l' unico mezzo di comunicazione tra gli SRO, che leggono e scrivono in essa i risultati.
Il meccanismo di focalizzazione consiste nell' uso di un set limitato di misure applicato a tutte le regioni (analisi globale) e nell' uso di misure piu' specialistiche e differenziate (analisi locale) applicate solo alle regioni candidate al riconoscimento da parte di un SRO. Lo scopo e' quello di ridurre i tempi di calcolo ed aumentare l' affidabilita' del riconoscimento.
La struttura di controllo e' costituita dal risolutore di conflitti, dall' interprete che attiva gli SRO e dalla sezione di controllo di terminazione.
Il risolutore di conflitti (ad es. una regione riconosciuta da 2 SRO diversi) riesamina le decisioni prese in precedenza, ed eventualmente fa rieseguire la segmentazione a livello locale o globale.
La sezione di controllo di terminazione attiva l' output dei risultati, costituito dalla mappa delle regioni riconosciute, quando nessuna altra regione puo' essere riconosciuta dal sistema (anche in caso di riconoscimento incompleto).
Il procedimento e' misto bottom-up top-down in quanto si opera talvolta basandosi sulle sole proprieta' intrinseche delle regioni, talvolta utilizzando il riconoscimento cosi' ottenuto per definire relazioni strutturali che si basano sulla conoscenza globale della sezione. Le relazioni strutturali vengono rappresentate mediante tabelle.
Tale schema di riconoscitore e' stato introdotto da M.Nagao e T.Matsuyama per il riconoscimento di immagini da ripresa aerea [4], ed e' stato opportunamente modificato per l' applicazione alle immagini NMR. Si e' curato in particolare il ruolo della conoscenza e lo sviluppo di una interfaccia uomo-macchina che permette di espandere con facilita' la conoscenza implementata nel sistema.
Sebbene attualmente venga eseguito il riconoscimento di una sola sezione del

cranio (sezione trasversale a livello oculare), il sistema e' predisposto all' estensione per il riconoscimento di diverse sezioni. A tale scopo e' stato definito un ulteriore livello di organizzazione della conoscenza: il sottosistema di riconoscimento di sezione (SRS). Per ciascuna sezione viene definito un set di organi caratteristici il cui riconoscimento attiva l' SRS corrispondente con i relativi SRO.

3. DISCUSSIONE

La complessita' della applicazione considerata sembra costituire un interessante banco di prova della validita' dell' impiego del linguaggio PROLOG in problemi di riconoscimento di immagini.
I risultati ottenuti, confrontati con quelli di una analoga implementazione in LISP, consentono di apprezzare la migliore leggibilita' del programma, fatto estremamente importante nel momento in cui si presenti la necessita' di effettuare delle modifiche; vanno inoltre sottolineati i tempi di sviluppo piu' ridotti. A fronte di tali vantaggi non si e' riscontrata una penalizzazione apprezzabile sui tempi di calcolo o sull' occupazione di memoria grazie ad un accorto uso di un ulteriore data base ad indirizzamento hash messo a disposizione dal sistema utilizzato. Opportune modifiche alle regole di produzione sono previste per poter rappresentare la conoscenza incerta mediante l' algebra dei fuzzy set.
Sviluppi futuri della ricerca possono consistere in:
- uso della mappa degli organi per identificare strutture 3D (eseguendo il riconoscimento di piu' sezioni contigue) e relativo display;
- misure automatiche sugli organi sia in 2D che in 3D;
- indagine sulla risposta dei vari tessuti nelle ampie condizioni di variabilita' dei segnali di eccitazione utilizzabili con l' NMR.

BIBLIOGRAFIA

1 Nillson N.J., "Principles of Artificial Intelligence", Springer Verlag, Berlin , 1982
2 Nandhakumar N. and Aggarwal J.K., "The Artificial Intelligence Approach to Pattern Recognition. A perspective and an overview", Pattern Recognition, vol. 18, n.6, pp. 383-389, 1985
3 Mansfield P. and Morris P.G., "NMR Imaging in Biomedicine", Academic Press , New York , 1982
4 Nagao M. and Matsuyama T., "Analysis of complex Aerial Photographs", Plenum, New York , 1980

INDICE DEGLI AUTORI