Teorie Formali

Una Teoria Formale e' una quadrupla

$T = \langle S, F, A, R \rangle$ dove

- S e' un insieme di simboli, una sequenza di simboli e' detta **espressione**
- F e' un sottoinsieme delle espressioni chiamato **formule ben formate (fbf)**
- A e' un sottoinsieme di F dette assiomi
- R e' un insieme finito di relazioni dette **regole di inferenza**

$$\frac{\textit{condizioni}}{\textit{conclusione}}$$

Una **dimostrazione** e' una sequenza finita di fbf $f_1, \dots, f_n$ tale che ciascuna f_i o e' un assioma o e' ricavabile dalle fbf precedenti mediante una regola di inferenza.

Un **teorema** e' l'ultima formula di una dimostrazione.

Data una formula F ed una teoria T , si dice che F e' **derivabile** da T ($T \vdash F$) se esiste una sequenza $f_1, \dots, f_n$ tale che $F = f_n$

Calcolo Proposizionale

Il Calcolo Proposizionale e' una Teoria Formale

Simboli:

- connettivi logici $\wedge, \vee, \neg, \equiv, \supset$
- le parentesi "(" e ")"
- un insieme numerabile di simboli proposizionali (asserzioni o variabili)

Formule ben formate (forme proposizionali) :

- tutti i simboli proposizionali sono fbf
- se V e W sono fbf allora lo sono $(\neg V), V \wedge W, V \vee W, V \equiv W, V \supset W$ sono fbf
- nessuna altra espressione e' una fbf

- Il calcolo proposizionale ha lo scopo di calcolare la **verita'** di una asserzione logica.
- Le fbf sono interpretate come asserzioni che possono essere **vere** o **false**.
- Ogni asserzione e' interpretata come una **variabile logica** che puo' assumere valore *vero* o *false*.
- Ad ogni fbf contenente le asserzioni $A_1, \dots, A_n$ e' associata una **funzione di verita'**

$$\{T, F\}^n \rightarrow \{T, F\}$$

- La funzione di verita' associata alle formule elementari contenenti un solo connettivo logico e' calcolata in base alle **tabelle di verita'**.
- La funzione di verita' associata ad una formula composta si ottiene combinando le funzioni di verita' dei componenti.

- Una forma proposizionale è una **tautologia** se la funzione di verità ad essa associata è sempre true.
- Una forma proposizionale è una **contraddizione** se la funzione di verità ad essa associata è sempre false.
- A è una contraddizione sse $\neg A$ è una tautologia.
- A **implica logicamente** B sse ogni assegnamento di valori di verità che rende vera A rende vera anche B .
- A e B sono **logicamente equivalenti** sse assumono sempre lo stesso valore di verità per ogni assegnamento di valori.

Equivalenze logiche

$A \wedge B$ e' logicamente equivalente a $\neg(A \supset \neg B)$

$A \wedge B$ e' logicamente equivalente a $\neg(\neg A \vee \neg B)$

$A \vee B$ e' logicamente equivalente a $\neg A \supset B$

$A \vee B$ e' logicamente equivalente a $\neg(\neg A \wedge \neg B)$

$A \equiv B$ e' logicamente equivalente a

$(A \supset B) \wedge (B \supset A)$

$\neg(A \vee B)$ e' logicamente equivalente a $\neg A \wedge \neg B$

$\neg(A \wedge B)$ e' logicamente equivalente a $\neg A \vee \neg B$

Una formula si dice in **forma normale**

coniuntiva (CNF) se e' una congiunzione di disgiunzioni.

Una formula si dice in **forma normale**

disgiuntiva (DNF) se e' una disgiunzione di congiunzioni.

Assiomi Logici

Siano A , B e C formule ben formate.

- A1. $A \supset (B \supset A)$
- A2. $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
- A3. $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$

Regola di inferenza: Modus Ponens

$$\frac{A, A \supset B}{B}$$

Il calcolo proposizionale e' una teoria formale
corretta (o **fondata**) e **completa**.

Corretta : tutto cio' che si riesce a dimostrare e'
vero.

Completa : tutto cio' che e' vero si puo'
dimostrare.

Calcolo dei predicati del Primo ordine

Definisce una classe di linguaggi. **Insieme dei simboli:**

- Insieme dei simboli di costante C
- Insieme dei simboli di variabile V
- Insieme dei simboli di funzione F
- Insieme dei simboli di predicato P
- I connettivi logici, $\wedge$, $\vee$, $\neg$, $\equiv$, $\supset$, le parentesi "(e)", i quantificatori esistenziale $\exists$ e universale $\forall$

Formule ben formate

Una **espressione o formula** e' una sequenza di simboli.

Un **termine** e' definito ricorsivamente come segue:

- una variabile e' un termine
- una costante e' un termine
- se f e' un simbolo di funzione n -aria e $t_1, \dots, t_n$ sono termini, allora $f(t_1, \dots, t_n)$ e' un termine.

Un **atomo o formula atomica** e' l'applicazione di un simbolo di predicato p n -ario ad n termini $t_1, \dots, t_n, p(t_1, \dots, t_n)$

Formule ben formate

Una **formula ben formata** e' definita ricorsivamente come segue:

- un atomo e' una fbf
- se A e B sono fbf allora lo sono anche $A \wedge B$, $A \vee B$, $\neg A$, $A \equiv B$, $A \supset B$
- se A e' una fbf e X e' una variabile allora $\forall X A$ e $\exists X A$ sono fbf.

Priorita' degli operatori:

$\neg, \forall, \exists$

$\wedge$

$\vee$

$\equiv, \supset$

- Un **letterale** e' un atomo o la negazione di un atomo.
- Una fbf si dice in **forma normale prenessa disgiuntiva (coniuntiva)** se e' la disgiunzione (coniunzione) di una o piu' fbf composte a loro volta da congiunzioni (disgiunzioni) di letterali e le quantificazioni compaiono tutte in testa.
- Il **campo di azione (scope)** di un quantificatore e' la formula che lo segue immediatamente

- Una variabile si dice **libera** se non compare nel campo di azione di un quantificatore.
- Le fbf che non contengono variabili libere si dicono **chiuse**.
- una formula che non contiene variabili si dice **ground**.
- Una formula F che si ottiene da un'altra formula G ridenominando le variabili si dice **variante** di G .

Interpretazioni

Una **interpretazione** definisce un insieme non vuoto D ed assegna

- a ciascun simbolo di costante in C una costante in D
- a ciascun simbolo di funzione n -ario in F una funzione

$$f : D^n \rightarrow D$$

- a ciascun simbolo di predicato in P una relazione in D^n

Si consideri un linguaggio in cui

$C = \{ "0" \}$, $F = \{ "s" \}$, s funzione unaria,

$P = \{ "p" \}$, p predicato binario

- Interpretazione I_1 : D insieme dei numeri naturali, la costante "0" rappresenta il numero zero, la funzione unaria s rappresenta il successore di un numero, p la relazione binaria $\leq$.
- Interpretazione I_2 : D insieme dei numeri interi negativi (incluso zero), la costante "0" rappresenta il numero zero, la funzione unaria s rappresenta il predecessore di un numero, p la relazione binaria $\leq$.
- Interpretazione I_3 : D insieme dei numeri interi negativi (incluso zero), la costante "0" rappresenta il numero zero, la funzione unaria s rappresenta il predecessore di un numero, p la relazione binaria $\geq$.

Una formula atomica ground ha valore vero sotto una interpretazione se il corrispondente predicato e' soddisfatto ed ha valore falso se il predicato corrispondente non e' soddisfatto.

- In I_1 la formula $p(0, s(0))$ e' vera, mentre $p(s(0), 0)$ e' falsa.
- In I_2 la formula $p(0, s(0))$ e' falsa, mentre $p(s(0), 0)$ e' vera.
- In I_3 la formula $p(0, s(0))$ e' vera, mentre $p(s(0), 0)$ e' falsa.

La verità di una formula composta ground si ottiene da quella delle sue componenti utilizzando le tavole di verità.

Una formula del tipo $\exists X F$ è vera in una interpretazione I se esiste almeno un elemento d del dominio tale che la formula ottenuta sostituendo d ad X è vera in I .

Esempio. La formula $\exists X, p(X, s(0))$ è vera in I_1 .

Una formula del tipo $\forall X F$ è vera in una interpretazione I se, per ogni elemento d del dominio, la formula ottenuta sostituendo d ad X è vera in I .

Esempio. La formula $\forall Y, p(0, Y)$ è vera in I_1 e I_3 , ed è falsa in I_2 .

- Una interpretazione I e' un **modello** per una formula ben formata chiusa F se e solo se F e' vera in I .
- Una fbf e' **soddisfacibile** se e solo se esiste almeno una interpretazione in cui e' vera.
- Una fbf e' **logicamente valida** se e solo se ha valore vero per tutte le possibili interpretazioni.
- Un insieme S di formule chiuse si dice **soddisfacibile** se e solo se esiste almeno una interpretazione in cui tutte le formule sono vere. Tale interpretazione e' detta **modello** per S .
- Un insieme S di formule chiuse si dice **insoddisfacibile** se e solo se non esiste nessuna interpretazione che soddisfi tutte le formule.

- Sia $S = \{p(s(0), 0) \supset p(0, s(0)), p(s(0), 0), \neg p(0, s(0))\}$ e' insoddisfacibile.
- Sia $S = \{\forall Y p(Y, Y), p(s(0), 0) \supset p(0, s(0))\}$. I_1 e I_3 sono modelli, I_2 non e' un modello.

- Date due formule A e B si dice che A **implica logicamente** B

$$A \models B$$

se ogni interpretazione che soddisfa A soddisfa anche B .

- A e B si dicono **logicamente equivalenti** se $A \models B$ e $B \models A$.

- Date una fbf F ed un insieme di formule S , F e' una **conseguenza logica** di S (segue logicamente da)

$$S \models F$$

se e solo se ogni interpretazione I che e' un modello per S e' anche un modello per F .

Esempio

Sia $S = \{p(0, 0), \forall X p(X, X), \forall X \forall Y p(X, Y) \supset p(X, s(Y))\}$.

In I_1 $0 \leq 0, \forall X X \leq X,$
 $\forall X \forall Y X \leq Y \supset X \leq s(Y))$

$p(0, s(0))$ segue logicamente da S ma non
 $p(s(0), 0)$.

Proprieta': Se una fbf F segue logicamente da un insieme S di formule, allora $S \cup \neg\{F\}$ e' insoddisfacibile. Se $S \cup \neg\{F\}$ e' insoddisfacibile (ed S era soddisfacibile) allora F segue logicamente da S .

Dimostrazione per assurdo: sistemi di refutazione

Sistemi assiomatico-deduttivi

Assiomi del Calcolo Predicativo

Siano A , B e C formule ben formate. Gli **assiomi logici** del calcolo dei predicati del primo ordine sono i seguenti:

- A1. $A \supset (B \supset A)$
- A2. $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
- A3. $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$
- A4. $\forall X A(X) \supset A(t)$ se $A(X)$ e' una fbf e t e' un termine che non contiene nessuna variabile libera Y che compare anche in $A(X)$ nel campo di azione di un quantificatore.
- A5. $\forall X (A \supset B) \supset (A \supset \forall X B)$ se A e' una fbf e X non compare libera in A .

Assiomi propri

Regole di inferenza:

Modus Ponens: $\frac{A, A \supset B}{B}$

Generalizzazione: $\frac{A}{\forall X A}$

Una **dimostrazione** e' una sequenza finita di fbf $f_1, \dots, f_n$ tale che ciascuna f_i o e' un assioma o e' ricavabile dalle fbf precedenti mediante una regola di inferenza.

Un **teorema** e' l'ultima formula di una dimostrazione.

Data una formula F ed una teoria T , si dice che F e' **derivabile** da T ($T \vdash F$) se esiste una sequenza $f_1, \dots, f_n$ tale che $F = f_n$ e, per ogni i , o f_i e' un assioma o e' ricavabile dalle fbf precedenti mediante una regola di inferenza.

Una teoria si dice **decidibile** se esiste un metodo meccanico per stabilire se una qualunque fbf e' un teorema oppure non lo e'.

Il calcolo dei predicati del primo ordine e' una teoria **semi-decidibile**, cioe', se una formula e' un teorema allora esiste un metodo meccanico per derivarla in un numero finito di passi. Se non e' un teorema, la procedura potrebbe non terminare.

Correttezza e Completezza

Un **modello** per una teoria T e' una interpretazione che soddisfa tutti gli assiomi (logici e propri) della teoria. Se T ha almeno un modello si dice **soddisfacibile**.

Una teoria assiomatica e' **corretta** se i teoremi dimostrati sono veri in tutti i modelli della teoria, cioe' seguono logicamente dalla teoria.

Una teoria assiomatica e' **completa** se tutte le fbf che sono vere in tutti i modelli della teoria possono essere dimostrati come teoremi della teoria.

$$T \vdash F \text{ se e solo se } T \models F$$

Clausole

Una **clausola** e' una disgiunzione di letterali

$$A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$$

Una clausola in cui non compare nessun letterale e' detta **clausola vuota** e si indica con un piccolo quadrato. La clausola vuota e' interpretata come *falso* $\vee$ $\neg$ *vero*, cioe' la contraddizione.

Ogni fbf del calcolo dei predicati del primo ordine si puo' trasformare in un insieme di clausole.

Proprieta': Sia T una teoria del primo ordine e T' una sua trasformazione in clausole. Allora T e' insoddisfacibile se e solo se T' e' insoddisfacibile.

Principio di Risoluzione

$$C_1 = A_1 \vee \dots \vee A_n$$

$$C_2 = B_1 \vee \dots \vee B_m$$

Se esistono un letterale A_i ed un letterale B_j tale che

$$A_i = A \text{ e } B_j = \neg A \text{ o viceversa}$$

dove A e' una formula atomica ground, allora si puo' derivare da C_1 e C_2 , dette clausole **parent**, una nuova clausola clausola C_3 , detta **risolvente**, della forma:

$$A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m$$

Unificazione

Una **sostituzione** e' un insieme di assegnamenti (legami) di termini a simboli di variabili.

$$\sigma = \{X_1/T_1, \dots, X_n/T_n\}$$

dove le X_i sono variabili distinte e $X_i \neq T_i$.

La sostituzione corrispondente all'insieme vuoto si chiama **sostituzione identita'** e si indica con ϵ .

Espressione: un termine, un letterale, una congiunzione o disgiunzione di letterali.

$[E]_\sigma$ indica l'applicazione della sostituzione σ all'espressione E ed e' detta **istanza** di E .

Dati $E = \{p(f(X, Z), f(g(Y)), a, Z), q(X, Y)\}$ e $\sigma = \{X/b, Y/f(a), Z/W\}$ allora

$$[E]_\sigma = \{p(f(b, W), f(g(f(a))), a, W), q(b, f(a))\}$$

Le sostituzioni si possono combinare tra di loro.

$$\sigma_1 = \{X_1/T_1, \dots, X_n/T_n\}$$

$$\sigma_2 = \{Y_1/Q_1, \dots, Y_m/Q_m\}$$

la **composizione** $\sigma_1\sigma_2$ e' la sostituzione ottenuta da

$$\{X_1/[T_1]_{\sigma_2}, \dots, X_n/[T_n]_{\sigma_2}, Y_1/Q_1, \dots, Y_m/Q_m\}$$

cancellando le coppie $X_i/[T_i]_{\sigma_2}$ per le quali

$$X_i = [T_i]_{\sigma_2}$$

e le coppie Y_j/Q_j per le quali

$$Y_j \in \{X_1, \dots, X_n\}$$

$$\sigma_1 = \{X/f(W), Y/Z, T/a\}$$

$$\sigma_2 = \{W/b, Z/Y, S/g(b), T/b\}$$

$$\sigma_1\sigma_2 = \{X/f(b), Y/Y, T/a, W/b, Z/Y, S/g(b), T/b\}$$

cancellando Y/Y e T/b si ha

$$\sigma_1\sigma_2 = \{X/f(b), T/a, W/b, Z/Y, S/g(b)\}$$

Per tutte le sostituzioni $\sigma_1, \sigma_2, \sigma_3$ e per qualunque espressione E valgono le seguenti proprietà:

$$\epsilon\sigma_1 = \sigma_1\epsilon = \sigma_1$$

$$(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$$

$$[[E]_{\sigma_1}]_{\sigma_2} = [E]_{(\sigma_1\sigma_2)}$$

Un insieme di atomi (o termini) $A_1, A_2, \dots, A_n$ e' **unificabile** se esiste una sostituzione σ tale che

$$[A_1]_\sigma = [A_2]_\sigma = \dots = [A_n]_\sigma$$

La sostituzione σ e' detta **sostituzione unificatrice (unificatore)**

$p(f(X), a)$ e $p(Y, f(W))$ non sono unificabili.

$p(f(X), Z)$ e $p(Y, a)$ sono unificabili e

$$\sigma = \{Y/f(a), X/a, Z/a\}$$

$p(a, X, f(g(Y)))$ e $p(Z, h(Z, W), f(W))$ sono unificabili e

$$\sigma = \{Z/a, X/h(a, g(b)), Y/b, W/g(b)\}$$

Un unificatore θ e' detto l' **unificatore piu' generale (mgu most general unifier)** se per ogni altro unificatore σ esiste una sostituzione γ tale che $\sigma = \theta\gamma$

Per l'esempio:

$$\theta = \{Y/f(X), Z/a\}$$

infatti $\sigma = \theta\{X/a\}$

$$\theta = \{Z/a, X/h(a, g(Y)), W/g(Y)\}$$

infatti $\sigma = \theta\{Y/b\}$

Teorema. Esiste un algoritmo che, dati due atomi, produce la loro sostituzione unificatrice piu' generale se essi sono unificabili, altrimenti esso termina in tempo finito restituendo come risultato la non esistenza di tale sostituzione.

Si definisce **insieme di non coincidenza** di un insieme di espressioni un insieme di sottoespressioni ottenuto come segue: si consideri ognuna delle espressioni e si individui la prima posizione in cui non tutte le espressioni hanno lo stesso simbolo. Si estrae allora da ogni espressione la sottoespressione che incomincia da quel simbolo.

$$S = \{p(f(X), h(Y), a), p(f(X), Z, a), p(f(X), h(Y), b)\}$$

l'insieme di non coincidenza e' $\{h(Y), Z\}$

$$S = \{$$

$$p(X, g(f(Y, Z), X), Y)$$

$$p(X, g(a, b), b),$$

$$p(X, g(g(h(X), a), Y), h(X))\}$$

l'insieme di non coincidenza

$$e' \{f(Y, Z), a, g(h(X), a)\}$$

Algoritmo di unificazione

Dato un insieme S di espressioni:

1. $k=0$, $\sigma_0 = \epsilon$
2. Se $[S]_{\sigma_k}$ e' una sola espressione allora **stop**, σ_k e' un mgu di S , altrimenti trova l'insieme di non coincidenza D_k di $[S]_{\sigma_k}$.
3. Se esistono v e t in D_k tale che v e' una variabile che non ricorre in t , si pone

$$\sigma_{k+1} = \sigma_k\{v/t\}$$

si incrementa k e si va al passo 2, altrimenti **stop**, S non e' unificabile.

Principio di Risoluzione per clausole generali

$$C_1 = A_1 \vee \dots \vee A_n$$

$$C_2 = B_1 \vee \dots \vee B_m$$

Se esistono un letterale $A_i = p(t_1, \dots, t_n)$ ed un letterale $B_j = p(l_1, \dots, l_n)$ tali che

$[p(t_1, \dots, t_n)]_\theta = \neg[p(l_1, \dots, l_n)]_\theta$ o viceversa dove θ e' la sostituzione unificatrice piu' generale, allora si puo' derivare da C_1 e C_2 , dette clausole **parent**, una nuova clausola clausola C_3 , detta **risolvente**, della forma:

$$[A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m]_\theta$$

$$C_1 = p(g(X), f(b)) \vee q(X, Z) \vee \neg s(f(X), b)$$

$$C_2 = \neg p(g(a), Y) \vee r(Y, Y)$$

$$\theta = \{X/a, Y/f(b)\}$$

$$C_3 = q(a, Z) \vee \neg s(f(a), b) \vee r(f(b), f(b))$$

Dimostrazione per contraddizione attraverso la risoluzione

Ricordiamo che una formula F segue logicamente da un insieme di formule H se e solo se $H \cup \neg\{F\}$ e' insoddisfacibile.

Dati gli assiomi propri H di una teoria ed una formula F , F e' un teorema della teoria se da $H \cup \neg\{F\}$ si deriva la contraddizione (clausola vuota) applicando la regola di risoluzione.

Teorema. Un insieme di clausole e' insoddisfacibile se e solo se l'algoritmo di risoluzione termina con successo in un numero finito di passi generando la clausola vuota.

La programmazione logica

Ricordiamo che una clausola e' una formula del tipo

$$A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$$

puo' essere riscritta come

$$B_1 \wedge \dots \wedge B_m \supset A_1 \vee \dots \vee A_n$$

sostituendo $\supset$ con $\leftarrow$, $\wedge$ con " , " e $\vee$ con " ; " si ha

$$A_1; \dots; A_n \leftarrow B_1, \dots, B_m$$

Una **clausola definita** e' una clausola che contiene un solo atomo positivo:

$$A \leftarrow B_1, \dots, B_m$$

$$A \leftarrow$$

Una **clausola negativa o clausola goal** e' una clausola che contiene solo atomi negativi:

$$\leftarrow B_1, \dots, B_m$$

$B_1, \dots, B_m$ sono chiamati sottogoal.

Clauseole di Horn

Una clausola di Horn e' una clausola definita o una clausola goal.

$A \leftarrow B_1, \dots, B_m$	regola
$A \leftarrow$	asserzione o fatto
$\leftarrow B_1, \dots, B_m$	goal
$\leftarrow$	clausola vuota o contraddizione

- Un **programma logico** e' un insieme di clausole definite (assiomi propri della teoria) su cui si vogliono fare delle interrogazioni (determinare se una formula e' un teorema della teoria).
- Una **computazione** corrisponde alla dimostrazione di una formula dal programma logico applicando il principio di risoluzione.
- Le formule da dimostrare possono essere solo congiunzioni di formule atomiche quantificate esistenzialmente.

Ricordiamo che

$$\neg \exists X A \equiv \forall X \neg A$$

$$\exists X \neg A \equiv \neg \forall X A$$

$$\neg(A_1 \wedge \dots \wedge A_n) \equiv \neg A_1 \vee \dots \vee \neg A_n$$

collega (maria,franco)

collega(luigi,franco)

collega(X,Z) $\leftarrow$ collega(X,Y),collega(Y,Z)

collega(X,Y) $\leftarrow$ collega(Y,X)

$\exists X (\text{collega}(\text{maria}, X) \wedge \text{collega}(X, \text{luigi}))$

$$\neg(\exists X \text{collega}(\text{maria}, X) \wedge \text{collega}(X, \text{luigi})) \equiv$$

$$\forall X \neg(\text{collega}(\text{maria}, X) \wedge \text{collega}(X, \text{luigi})) \equiv$$

$$\forall X \neg \text{collega}(\text{maria}, X) \vee \neg \text{collega}(X, \text{luigi}) \equiv$$

$$\leftarrow \text{collega}(\text{maria}, X), \text{collega}(X, \text{luigi}) \equiv$$

Risoluzione SLD

Risoluzione Lineare per clausole Definite con funzione di Selezione

- Dati un programma logico P ed una clausola goal G_0 , la risoluzione *SLD* cerca di derivare la contraddizione logica da $P \cup \{G_0\}$.
- Al generico passo i , dalla clausola goal G_i e da una clausola C_i del programma P , si cerca di derivare il nuovo risolvente G_{i+1} .
- La risoluzione SLD seleziona un atomo A_m dal goal G_i e lo unifica con la testa della clausola C_i attraverso la sostituzione unificatrice piu' generale θ_i .
- il nuovo risolvente G_{i+1} si ottiene sostituendo l'atomo selezionato con il corpo della clausola C_i applicando l'mgu θ_i .

$$G_i : \leftarrow A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$$

$$C_i : A \leftarrow B_1, \dots, B_q$$

tali che

$$[A_m]_{\theta_i} = [A]_{\theta_i}$$

allora la risoluzione *SLD* deriva

G_{i+1} :

$$\leftarrow [A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k]_{\theta_i}$$

Una **derivazione** *SLD* per un goal G_0
dall'insieme di clausole definite P e' una sequenza
di clausole goal

$$G_0, \dots, G_n$$

una sequenza di varianti di clausole del programma

$$C_1, \dots, C_n$$

ed una sequenza di sostituzioni mgu

$$\theta_1, \dots, \theta_n$$

tali che G_{i+1} e' derivato da G_i e C_{i+1} usando θ_{i+1}

Si hanno tre possibili tipi di derivazioni:

- **successo**, se per n finito si ha che $G_n = \leftarrow$
(clausola vuota)
- **fallimento finito**, se per n finito non e' piu' possibile derivare da G_n alcun risolvente e $G_n \neq \leftarrow$
- **infinita**, se e' sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.

Una **refutazione SLD** e' una derivazione SLD di successo.

- L' **insieme di successo** di un programma logico P e' l'insieme di tutti gli atomi ground A tali che $P \cup \neg\{A\}$ ha una refutazione SLD.
- Dati un programma logico P ed una clausola goal G_0 , una **risposta calcolata** θ per $P \cup \{G_0\}$ e' la sostituzione ottenuta restringendo le composizione delle sostituzioni mgu $\theta_1, \dots, \theta_n$ utilizzate nella refutazione SLD di $P \cup \{G_0\}$ alle variabili di G_0 .
- Una **risposta corretta** per $P \cup \{G_0\}$ e' una sostituzione θ tale che se G_0 e'
 $\leftarrow A_1, \dots, A_k$
 allora la formula

$$F = [A_1 \wedge \dots \wedge A_k]_{\theta}$$

e' una conseguenza logica di P , cioe'
 $P \cup \{[G_0]_{\theta}\}$ e' insoddisfacibile.

Correttezza e completezza della risoluzione SLD

La risoluzione SLD e' **corretta**, cioe' ogni risposta calcolata per $P \cup \{G_0\}$ e' una risposta corretta per $P \cup \{G_0\}$.

La risoluzione SLD e' **completa**, cioe', se σ e' una risposta corretta per $P \cup \{G_0\}$, allora esiste una risposta calcolata per $P \cup \{G_0\}$ ed una sostituzione λ tale che $\sigma = \theta\lambda$.

La sostituzione λ stabilisce il legame tra risposta corretta e calcolata in quanto la risposta calcolata puo' essere piu' generale di quella corretta.

Regola di Calcolo

Una **regola di calcolo** e' una funzione che, dato un goal,

$$\leftarrow A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$$

seleziona un suo atomo A_m , detto **atomo selezionato**.

Dati un programma logico P , una clausola goal G_0 ed una regola di calcolo R , una **derivazione SLD** di $P \cup \{G_0\}$ *via* R e' una derivazione *SLD* di $P \cup \{G_0\}$ nella quale si usa la funzione R per determinare l'atomo selezionato ad ogni passo di risoluzione.

Refutazione SLD di $P \cup \{G_0\}$ *via* R : si definisce in modo analogo.

Proprieta'. (*Indipendenza dalla regola di calcolo*) Dato un programma logico P , l'insieme di successo di P non dipende dalla regola di calcolo utilizzata dalla risoluzione SLD.

Alberi SLD

Dati un programma logico P , una clausola goal G_0 ed una regola di calcolo R , un **albero SLD** $P \cup \{G_0\}$ via R e' definito come segue:

- Ogni nodo dell'albero e' un goal;
- la radice dell'albero e' il goal G_0 ;
- il nodo vuoto (indicato con $\leftarrow$) non ha figli;

- dato il nodo

$$\leftarrow A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$$

se A_m e' l'atomo selezionato dalla regola di calcolo R , allora questo nodo (**genitore**) ha un nodo **figlio** per ciascuna clausola

$$C = A \leftarrow B_1, \dots, B_q$$

tale che A ed A_m sono unificabili attraverso una sostituzione unificatrice piu' generale θ . Il nodo figlio e' etichettato con la clausola goal

$$\leftarrow [A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k]_{\theta_i}$$

ed il ramo dal nodo padre al nodo figlio e' etichettato con la sostituzione θ e la clausola selezionata C .

Strategia di ricerca

La strategia di ricerca stabilisce come esplorare l'albero di risoluzione *SLD* per cercare i nodi di successo.

- Ricerca in profondità' (depth first) con backtracking
- Ricerca in ampiezza (breadth first)

Interpretazione Procedurale

Una **dichiarazione di procedura** e' un insieme di clausole definite aventi lo stesso simbolo di predicato come testa.

$$A \leftarrow B_1, \dots, B_n$$

A rappresenta l'**intestazione della procedura** e $B_1, \dots, B_n$ il **corpo**.

- Il **nome della procedura** e' dato dal simbolo di predicato di testa piu' la sua arita' (numero di argomenti).
- I **parametri formali** della procedura sono i termini che compaiono nella testa delle clausole che definiscono la procedura.
- Le procedure possono essere **non deterministiche**: ad uno stesso nome sono associate piu' definizioni.

- La **chiamata di procedura** e' realizzata da una clausola goal.
- I **parametri attuali** sono i termini che compaiono nella chiamata della procedura.
- Una variabile nella intestazione di procedura e' di ingresso o di uscita in base alla chiamata (**reversibilita'** o **invertibilita'**).
- La clausola vuota rappresenta la terminazione con successo della procedura.

Il linguaggio Prolog

Un programma Prolog e' un insieme di clausole definite della forma:

A. (**fatto**)

A : $\neg B_1, \dots, B_n$. (**regola**)

A e B_i sono formule atomiche.

Una formula atomica e' una formula del tipo

$$p(t_1, \dots, t_n)$$

dove p e' un simbolo di predicato e $t_1, \dots, t_n$ sono termini.

Un **termine** e' definito ricorsivamente come segue:

- una costante e' un termine. In Prolog le costanti sono i numeri (interi e floating point) e gli atomi alfanumerici (il primo carattere deve essere un carattere alfabetico minuscolo);
- una variabile e' un termine. Le variabili in Prolog sono stringhe alfanumeriche aventi come primo carattere un carattere alfabetico maiuscolo o il carattere `_`.
- se f e' un simbolo di funzione n-aria e $t_1, \dots, t_n$ sono termini, allora $f(t_1, \dots, t_n)$ e' un termine. In Prolog $f(t_1, \dots, t_n)$ prende il nome di **struttura**.

Un goal (o query) ha la forma :

$?-B_1, \dots, B_n.$

Interpretazione Dichiarativa

Le variabili che occorrono all'interno di una clausola sono considerate quantificate universalmente.

- Dato un fatto

$$p(t_1, \dots, t_m)$$

se $X_1, \dots, X_n$ sono le variabili che compaiono in $t_1, \dots, t_m$, il significato è il seguente:

$$\forall X_1, \dots, \forall X_n (p(t_1, \dots, t_m))$$

- data una regola

$$A : \neg B_1, \dots, B_k.$$

se $Y_1, \dots, Y_m$ sono le variabili che compaiono solo nel corpo della regola e $X_1, \dots, X_n$ sono le variabili che compaiono sia nel corpo che nella testa della regola, il significato e' il seguente:

$$\forall X_1, \dots, \forall X_n, \forall Y_1, \dots, \forall Y_m (B_1 \wedge \dots \wedge B_k \supset A)$$

ovvero

$$\forall X_1, \dots, \forall X_n, \forall Y_1, \dots, \forall Y_m (\neg(B_1 \wedge \dots \wedge B_k) \vee A)$$

ovvero

$$\forall X_1, \dots, \forall X_n, (\neg(\exists Y_1, \dots, \exists Y_m (B_1 \wedge \dots \wedge B_k)) \vee A)$$

ovvero

$$\forall X_1, \dots, \forall X_n, ((\exists Y_1, \dots, \exists Y_m (B_1 \wedge \dots \wedge B_k)) \supset A)$$

- Dato un programma P ed un goal (o query)

$$?-p(t_1, \dots, t_m)$$

se $X_1, \dots, X_n$ sono le variabili che compaiono in $t_1, \dots, t_m$, il significato della query e' il seguente:

$$\exists X_1, \dots, \exists X_n p(t_1, \dots, t_m)$$

L'obiettivo e' di calcolare una sostituzione

$$\sigma = \{X_1/s_1, \dots, X_n/s_n\}$$

per cui

$$P \vdash [p(t_1, \dots, t_m)]_\sigma$$

Regola di calcolo

La regola di calcolo utilizzata dal Prolog e' la **left-most**. Dato un goal

?- $G_1, \dots, G_n$.

viene selezionato sempre l'atomo piu' a sinistra.

Strategia di ricerca

La strategia di ricerca utilizzata dal Prolog e' una **strategia di ricerca in profondita'** (depth-first) con backtracking cronologico

Dato un goal

$$? - G_1, G_2, \dots, G_n.$$

sia G_1 il letterale da risolvere e

$$A : -B_1, \dots, B_k.$$

la prima clausola la cui testa A unifica con G_1 . Il goal viene ridotto a :

$$? - [B_1, \dots, B_k, G_2, \dots, G_n]_{\sigma}.$$

Occur check nell'algoritmo di unificazione non eseguito.

Aritmetica

In Prolog e' possibile costruire i numeri naturali a partire dal simbolo di costante "0" ed il simbolo di funzione unario "s" (successore).

Le operazioni sui numeri naturali sono facilmente definite nel seguente modo:

(1) somma (0,X,X).

(2) somma(s(X),Y,s(Z)):- somma(X,Y,Z).

Dato il goal

?-somma(s(0),s(0),W).

Il goal unifica con la clausola (2) quando $X=0$,
 $Y=s(0)$, $W=s(Z)$.

Il nuovo goal diventa:

?-somma(0,s(0),Z).

che unifica con la clausola (1) quando
 $X1=s(0)$, $Z=X1$.

Quindi la soluzione e' $W=s(Z)=s(s(0))$.

(1) prodotto (0,X,0).

(2) prodotto(s(X),Y,Z):-

prodotto(X,Y,W),somma(W,Y,Z).

Dato il goal

?-prodotto(s(s(0)),s(s(0)),T).

la risoluzione procede come segue:

unificazione con la clausola (2) con sostituzione

$X1=s(0), Y1=s(s(0)), T=Z1$

**?-prodotto(s(0),s(s(0)),W1),
somma(W1,s(s(0)),Z1).**

unificazione con la clausola (2) con sostituzione

$X2=0, Y2=s(s(0)), W1=Z2$

**?-prodotto(0,s(s(0)),W2),
somma(W2,s(s(0)),Z2),
somma(Z2,s(s(0)),Z1).**

unificazione con la clausola (2) con sostituzione

$X3=s(s(0)), W2=0$

**?-somma(0,s(s(0)),Z2),
somma(Z2,s(s(0)),Z1).**

unificazione con la clausola (1) con sostituzione
 $Z2=s(s(0))$

?-somma(s(s(0)),s(s(0)),Z1).

risolvendo questo goal si ha $Z1=s(s(s(0)))$ e
quindi $T=Z1=s(s(s(0)))$

fattoriale(0,s(0)).

fattoriale(s(N),F) :-

fattoriale(N,F1),prodotto(s(N),F1,F).

Operatori aritmetici

exp, log, ln, sin, cos, tg

*+, -, *, /, mod, div*

Operatori relazionali

<, >, <=, >=, :=, = / =

Predicato is

T is Expr

dove *T* può essere un atomo numerico o una variabile

Expr deve essere una espressione.

Significato: *Expr* viene valutata ed il risultato della valutazione viene unificato con *T*.

Liste

Una lista é definita come segue:

- l'atomo `[]` é una lista (la lista vuota);
- il termine `.(T,LISTA)` é una lista se T é un termine e LISTA é una lista. T prende in nome di **testa (head)** della lista e LISTA di **coda (tail)** della lista.

Esempi:

`[]`

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

`.(.(f(a),[]),.(c,[ ]))`

Altra notazione:

`.(T,LISTA)` si puó scrivere `[T|LISTA]`

`[a | [b|[c|[e| [ ]]]]]`

`[[f(a)|[ ]][c|[ ]]]`

Ulteriore semplificazione:

$$[a | [b|[c|[e| []]]]] = [a,b,c,e]$$

$$[[f(a)|[]][c|[]]] = [[f(a)],c]$$

Unificazione sulle liste

$$[a,[b,c],d,[e,f,[g,h]],k]$$

$$[X|Y] : X=a, Y=[[b,c],d,[e,f,[g,h]],k]$$

$$[X,Y|Z] : X=a, Y=[b,c], Z=[d,[e,f,[g,h]],k]$$

$$[-,-,-,[-,X|Y]|Z] : X=f, Y=[[g,h]], Z=k$$

Operazioni sulle liste

Verificare se un elemento appartiene ad una lista

`member(X,[X|T]).`

`member(X,[Y|T]):-member(X,T).`

Lunghezza di una lista

`length([],0).`

`length([X|T],N):-length(T,N1), N is N1 + 1.`

definizione alternativa:

`length1(L,N):-length1(L,0,N).`

`length1([X|T],Acc,N):- Acc1 is Acc + 1,
length1(T,Acc1,N).`

`length1([],Acc,Acc).`

Concatenazione di due liste

`append([ ],L,L).`

`append([H|T],L,[H|T1]) :- append(T,L,T1).`

Ultimo elemento di una lista

`ultimo(X,[X]).`

`ultimo(X,[_|T]):-ultimo(X,T).`

Inversione di una lista

`reverse([ ],[ ]).`

`reverse([X|T],Z) :- reverse(T,T1),`

`append(T1,[X],Z).`

definizione alternativa

`reverse(X,Y):- reverse1(X,[ ],Y).`

`reverse1([X|T],Acc,Y):- reverse1(T,[X|Acc],Y).`

`reverse1([ ],Y,Y).`

Sottoliste

$\text{prefix}(\text{Prefisso}, \text{Lista})$: é vero se Prefisso é una sottolista iniziale di Lista

$\text{prefix}([], \text{Ys})$.

$\text{prefix}([X|\text{Xs}], [X|\text{Ys}])$:- $\text{prefix}(\text{Xs}, \text{Ys})$.

$\text{suffix}(\text{Suffisso}, \text{Lista})$: é vero se Suffisso é una sottolista terminale di Lista

$\text{suffix}(\text{Xs}, \text{Xs})$.

$\text{suffix}(\text{Xs}, [Y|\text{Ys}])$:- $\text{suffix}(\text{Xs}, \text{Ys})$.

Una sottolista puó essere definita come:

$\text{sublist}(\text{Xs}, \text{Ys})$:- $\text{prefix}(\text{Xs}, \text{Ys})$.

$\text{sublist}(\text{Xs}, [Y|\text{Ys}])$:- $\text{sublist}(\text{Xs}, \text{Ys})$.

Esercizio

Scrivere un programma Prolog che, data una lista L di interi, verifichi se esiste una sottolista $L1$ tale che la somma degli elementi di $L1$ sia la metà della somma degli elementi di L . Ad esempio se $L=[1,5,10,5,10,9]$, la sottolista $[5,10,5]$ verifica la proprietà richiesta.

```
verifica(L,L1):- sublist(L1,L),  
                  somma(L,K),  
                  somma(L1,K1), K1 is K/2.
```

Ordinamento di Liste

Permutation sort:

```
sort(Xs,Ys):-permutation(Xs,Ys),ordered(Ys).
```

```
permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys),  
                        permutation(Ys,Zs).
```

```
permutation([ ],[ ]).
```

```
ordered([X]).
```

```
ordered([X,Y|Ys]):-X=<Y,ordered([Y|Ys]).
```

```
select(X,[X|Xs],Xs).
```

```
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

Insertion sort

`sort([X|Xs],Ys):- sort(Xs,Zs),insert(X,Zs,Ys).`

`sort([],[]).`

`insert(X,[],[X]).`

`insert(X,[Y|Ys],[Y|Zs]):-X > Y, insert(X,Ys,Zs).`

`insert(X,[Y|Ys],[X,Y|Ys]):- X=< Y.`

Quicksort

```
quicksort([X|Xs],Ys):- partition(Xs,X,Littles,Bigs),  
                        quicksort(Littles,Ls),  
                        quicksort(Bigs,Bs),  
                        append(Ls,[X|Bs],Ys).
```

```
quicksort([],[]).
```

```
partition([X|Xs],Y,[X|Ls],Bs):-X=<Y,  
                                partition(Xs,Y,Ls,Bs).
```

```
partition([X|Xs],Y,Ls,[X|Bs]):-X>Y,  
                                partition(Xs,Y,Ls,Bs).
```

```
partition([],_,[],[]).
```

Scrivere un programma Prolog che, date tre liste, costruisca la lista degli elementi che compaiono in almeno due delle tre liste. Ad esempio date $L_1 = [1, 3, 5, 7]$, $L_2 = [2, 3, 6, 9]$ e $L_3 = [10, 3, 5, 1]$, gli elementi che appaiono in almeno due di queste tre liste sono $[1, 3, 5]$.

```

crea(L1,L2,L3,L):-intersezione(L1,L2,B1),
    intersezione(L1,L3,B2), intersezione(L2,L3,B3),
    append(B1,B2,B4),append(B4,B3,B5),
delduplicati(B5,L).

intersezione([X|T],L,[X|Ts]):-member(X,L),
    intersezione(T,L,Ts).

intersezione([X|T],L,Ts):- intersezione(T,L,Ts).
intersezione([ ],L,[ ]).

delduplicati([X|T],[X|T1]):-delall(X,T,L1),
    delduplicati(L1,T1).

delduplicati([ ],[ ]).

delall(-, [ ], [ ]).

delall(X, [X|L], L1) :- delall(X, L, L1).

delall(X, [Y|L], [Y|L1]) :- X\==Y, delall(X, L,L1).

```

Il predicato CUT (!)

$p : \neg q_1, q_2, \dots, q_i!, q_{i+1}, q_{i+2}, \dots, q_n.$

se q_{i+1} fallisce non é possibile cercare una soluzione alternativa per $q_1, q_2, \dots, q_i$ e p .

Consideriamo il programma:

1. g:-a.

2. g:-s.

3. a:-p,b.

4. a:-r.

5. p:-q.

6. p:-r.

7. q.

8. r.

?- g.

Cosa succede se inseriamo il cut nella regola 3:

a:-p!,b.

Green CUT - Red CUT

Green CUT: la sua introduzione non influenza il significato del programma, l'insieme delle soluzioni é lo stesso ma lo spazio di ricerca é ridotto.

Red CUT: la sua introduzione cambia il significato del programma e l'insieme delle soluzioni non é lo stesso.

sumto(N,Ris): somma i numeri da 1 ad N e restituisce il risultato in Ris.

sumto(1,1):-!

sumto(N,Ris):-N1 is N - 1, sumto(N1,Ris1),
Ris is Ris1+1.

?- sumto(3,X). yes, X=6.

?- sumto(1,X). yes, X=1;

il programma, se non ci fosse il cut, andrebbe in loop.

Specifica del determinismo

$p:-a,b.$

$p:-c.$

if a then b else c : $p:-a,!b.$

$\text{delete}(X, [], []).$

$\text{delete}(X, [X|L], L) :- !.$

$\text{delete}(X, [Y|L], [Y|L1]) :- \text{delete}(X, L, L1).$

$\text{delall}(-, [], []).$

$\text{delall}(X, [X|L], L1) :- !, \text{delall}(X, L, L1).$

$\text{delall}(X, [Y|L], [Y|L1]) :- \text{delall}(X, L, L1).$

Efficienza

`merge(L1,L2,L)`: L é la lista ordinata ottenuta dalla fusione di L1 ed L2 (giá ordinate).

`merge([X|L1],[Y|L2],[X|L])`:-X= $\leq$ Y,!,
 `merge(L1,[Y|L2],L)`.

`merge([X|L1],[Y|L2],[Y|L])`:-X > Y,!,
 `merge([X|L1],L2,L)`.

`merge([ ],L,L)`.

`merge(L,[ ],L)`.

Il predicato FAIL

Predicato predefinito che, quando viene valutato, fallisce.

```
itera:- p(X),verifica(X),fail.
```

```
itera.
```

Il predicato CALL

call(T) : il termine T é considerato un predicato e viene valutato dall'interprete.

p(X):- call(X).

q(a).

?- p(q(a)).

yes.

?- p(q(X)).

yes, X=a.

La negazione in Prolog

not P : viene valutato P e, se la dimostrazione di P fallisce in modo finito, not P ha successo.

p:-q,r.

p:-t.

q:-a.

r.

not(P):- call(P), !, fail.

not(P).

?- not p(X).

Il significato é : $\exists X \text{ not } p(X)$. In Prolog si esegue prima p(X) e poi si applica il not cioè:

?- p(X). ($\exists X p(X)$)

not ($\exists X p(X)$) $\equiv \forall X \text{ not } p(X)$.

celibe(X):- not sposato(X), adulto(X).

sposato(luigi).

adulto(marco).

?- celibe(X).

no.

Negazione safe : quando vengono valutati gli atomi devono essere ground.

SETOF e BAGOF

setof(X,P,S) ordina ed elimina i duplicati

bagof(X,P,S)

S é l'insieme delle istanze di X tale che P sia soddisfatto.

amico(luigi,paolo). amico(mario,carlo).

amico(giovanni,paolo). amico(marco,carlo).

amico(tommaso,paolo). amico(tommaso,carlo).

?- setof(X,amico(X,Y),S).

Y=paolo, S=[giovanni,luigi,tommaso];

Y=carlo, S=[marco,mario,tommaso].

?- setof((Y,S),setof(X,amico(X,Y),S),SS).

SS=[(carlo,[marco,mario,tommaso])

(paolo,[giovanni,luigi,tommaso])].

setof($X, Y \wedge P, S$): S é l'insieme delle istanze di X
per cui esiste un Y per cui P é soddisfatto.
?- setof($X, Y \wedge \text{amico}(X, Y), S$).
 $S = [\text{marco}, \text{mario}, \text{giovanni}, \text{luigi}, \text{tommaso}]$.

Manipolazione di termini

Confronto di espressioni aritmetiche:

- $E1 ::= E2$ i valori delle due espressioni $E1$ ed $E2$ sono uguali.
- $E1 \neq E2$ i valori delle due espressioni $E1$ ed $E2$ sono diversi.
- $E1 > E2$ il valore dell'espressione $E1$ é maggiore del valore dell'espressione $E2$.
- $E1 \geq E2$ il valore dell'espressione $E1$ é maggiore o uguale del valore dell'espressione $E2$.
- $E1 < E2$ il valore dell'espressione $E1$ é minore del valore dell'espressione $E2$.
- $E1 \leq E2$ il valore dell'espressione $E1$ é minore o uguale del valore dell'espressione $E2$.

Confronto di termini:

- $X = Y$ verifica se X e Y sono unificabili.
- $X == Y$ verifica se X e Y sono identici.
- $X \neq Y$ verifica se X e Y non sono unificabili.

$X \neq Y$:- $X=Y,!,fail$.

$X \neq Y$.

- $X \neq Y$ verifica se X e Y non sono uguali.

$X \neq Y$:- $X=Y,!,fail$.

$X \neq Y$.

Classificazione di termini

- $\text{var}(X)$: ha successo se X é una variabile non istanziata
- $\text{nonvar}(X)$: X non é una variabile
- $\text{atom}(X)$: X é un atomo non numerico
- $\text{number}(X)$: X é un numero
- $\text{integer}(X)$: X é un numero intero
- $\text{atomic}(X)$: X é un atomo oppure un numero

$\text{atomic}(X):-\text{atom}(X).$

$\text{atomic}(X):-\text{number}(X).$

Accesso alle componenti di un termine

- $\text{functor}(\text{Term}, \text{Functor}, \text{Arity})$

Term é una struttura con funtore Functor e numero di argomenti Arity.

?- $\text{functor}(g(a,b,f(Z)), F, A)$.

Z= -, F=g, A=3.

?- $\text{functor}(T, f, 2)$.

T=f(-,-)

- $\text{arg}(\text{Pos}, \text{Term}, \text{Arg})$

Arg é l'argomento di Term in posizione Pos:

Pos e Term devono essere sempre istanziati

?- $\text{arg}(2, f(a, g(b)), X)$.

X=g(b).

?- $\text{arg}(2, [a, b, c], X)$.

X=b.

- il predicato UNIV =..

T =.. L

L é la lista la cui testa contiene il funtore del termine T ed il resto degli elementi sono gli argomenti di T.

?- f(a,b,c) =..[f,a,b,c].

yes.

?- f(a,b,c) =.. L.

yes, L=[f,a,b,c].

?- X =.. [f,a,h(c)].

yes, X=f(a,h(c)).

?- X =.. [append,[a,b,c],[d,e],[a,b,c,d,e]].

yes, X=append([a,b,c],[d,e],[a,b,c,d,e]).

STRINGHE

name(A,L) : L é la lista dei caratteri ASCII
corrispondenti all'atomo A.

?- name(abc,L).

L=[97,98,99]

ciaoada(C):-name(hello,L),name(ada,H),
append(L,[32|H],T),name(C,T).

?- ciaoada(T).

T = 'hello ada'

Utilizzo di functor, arg e univ

subterm(Sub,Term) : Sub é un sottoterminine del termine Term.

subterm(Term,Term).

subterm(Sub,Term):-functor(Term,F,N),
subterm1(N,Sub,Term).

subterm1(N,Sub,Term):- N > 1, N1 is N-1,
subterm1(N1,Sub,Term).

subterm1(N,Sub,Term):- arg(N,Term,Arg),
subterm(Sub,Arg).

Definizione alternativa di subterm:

subterm(Term,Term).

subterm(Sub,Term):-Term =.. [F|Args],
 subtermlist(Sub,Args).

subtermlist(Sub,[A|Args]):- subterm(Sub,A).

subtermlist(Sub,[A|Args]):- subtermlist(Sub,Args).

ground(Term) : ha successo se Term é un termine
ground.

ground(Term):- nonvar(Term), atomic(Term).

ground(Term):- nonvar(Term), functor(Term,F,N),
 ground(N,Term).

ground(N,Term):- N > 0, arg(N,Term,Arg),
 ground(Arg), N1 is N-1, ground(N1,Term).

ground(0,Term).

Predicati di Metalivello

- accesso alle clausole di un programma
- modificare dinamicamente un programma
- meta-programmazione

`clause(Head,Body)`: Head é una clausola il cui corpo é Body, Head deve essere istanziato.

`member(X,[X|T])`.

`member(X,[Y|T]):-member(X,T)`.

?- `clause(member(X,L),B)`.

ha due soluzioni (1) $L = [X|T]$ e $B = \text{true}$ e

(2) $L = [Y|T]$ e $B = \text{member}(X,T)$

Modificare dinamicamente il database

- `assert(Clausola)` aggiunge Clausola al programma
`asserta(Clausola)` Clausola é aggiunta all'inizio
`assertz(Clausola)` Clausola é aggiunta alla fine
- `retract(Clausola)` Clausola é eliminata dal programma
- `abolish(Nome, Arity)` elimina tutte le clausole con predicato Nome e numero di argomenti Arity.

Siano dati un insieme di fatti $p(X)$, dove X é un numero. Scrivere un programma Prolog *somma(N)* che restituisca in N la somma degli X . Ad esempio, se nel database sono memorizzati $p(2), p(10), p(3)$, allora $N=15$.

```
somma(N) :- somma(0,N),ripristina.
somma(T,N) :- p(X),N1 is T + X,
    retract(p(X)),assert(p1(X)),somma(N1,N).
somma(N,N).
ripristina :- p1(X),assert(p(X)),
    retract(p1(X)),ripristina.
ripristina.
```

Soluzione alternativa con fail:

```
somma(N) :- assert(somma1(0)),
    calcola, somma1(N).
calcola :- p(X),somma1(N),N1 is N + X,
    retract(somma1(N)),assert(somma1(N1)),fail.
calcola.
```

Meta-interpretate

```
solve(true).  
solve((A,B)) :- solve(A),solve(B).  
solve(A) :- clause(A,B), solve(B).  
system(A is B).  
system(atom(X)).  
system(clause(A,B)).  
:  
solve(A) :- system(A), A.  
solve(not A) :- not solve(A).  
solve(setof(X,Goal,L)):-setof(X,solve(Goal),L).
```

Alberi

Un albero é una struttura dati definita come segue:

- l'albero vuoto (denotato con *nil*) é un albero;
- un nodo N (detto radice) cui sono associati $n \geq 0$ alberi, detti sottoalberi di N , é un albero.

In Prolog gli alberi possono essere rappresentati in tre possibili modi:

- come termine strutturato;
- mediante liste;
- come insiemi di fatti.

Visita di Alberi binari

$f(T1, T2)$: nodo con sottoalbero sinistro $T1$ e destro $T2$.

$f(T1, nil)$: nodo con sottoalbero sinistro $T1$ e destro vuoto.

$f(nil, T2)$: nodo con sottoalbero sinistro vuoto e sottoalbero destro $T2$.

$f(nil, nil)$: nodo foglia.

- pre-order: viene visitata la radice, quindi il sottoalbero sinistro e poi quello destro.
- in-order : viene visitato il sottoalbero sinistro, quindi la radice, infine il sottoalbero destro.
- post-order: viene visitato il sottoalbero sinistro, poi quello destro e infine la radice.

Visita in-order di un albero binario

Rappresentazione come termine strutturato.

```
inorder(nil).
```

```
inorder(Tree) :- functor(Tree,Node,2),  
    arg(1,Tree,LeftSubtree), inorder(LeftSubtree),  
    visita(Node),  
    arg(2,Tree,RightSubtree), inorder(RightSubtree).
```

Rappresentazione con le liste.

```
inorder(nil).
```

```
inorder([Nodo,Sinistro,Destro]):- inorder(Sinistro),  
    visita(Nodo), inorder(Destro).
```

Rappresentazione come insieme di fatti:

```
inorder(nil).
```

```
inorder(Nodo):-  
    bagof(Figlio,f(Nodo,Figlio),[Sinistro,Destro]),  
    inorder(Sinistro),visita(Nodo),inorder(Destro).
```

Scrivere un programma prolog che dato un albero binario di ricerca T contenente numeri interi e dati due valori interi I e J restituisca il sottoalbero minimo contenente entrambi i nodi. Si consideri la rappresentazione a liste [Radice,SottoAlbSinistro, SottoAlbDestro]

```
mintree(I,J,Tree,MinT):-visitaT(I,Tree,L1),
                        visitaT(J,Tree,L2),
                        intersect(L1,L2,Radice),
                        sottoalbero(Radice,Tree,MinT).
```

```
visitaT(I,[I,S,D],[I]).
```

```
visitaT(I,[Radice,S,D],[Radice|L]):-I<Radice,
                                       visitaT(I,S,L).
```

```
visitaT(I,[Radice,S,D],[Radice|L]):-I>Radice,
                                       visitaT(I,D,L).
```

```

intersect([X,Y|L1],[X,Z|L2],X):-Z\==Y.
intersect([X,Y],[X],X).
intersect([X],[X,Y],X).
intersect([X,Y|L1],[X,Y|L2],Radice):-
    intersect([Y|L1],[Y|L2],Radice).
sottoalbero(Radice,[Radice,S,D],[Radice,S,D]).
sottoalbero(Radice,[Radice1,S,D],R):-
    Radice<Radice1,
    sottoalbero(Radice,S,R).
sottoalbero(Radice,[Radice1,S,D],R):-
    Radice>Radice1,
    sottoalbero(Radice,D,R).

```

Scrivere un predicato Prolog *conta*(*Tree*, *N*) che restituisca in *N* il numero di nodi di un albero binario ed un predicato *distinti*(*Tree*, *N*) che restituisca in *N* il numero di informazioni distinte contenute nell'albero. L'albero sia rappresentato mediante un termine complesso del tipo *t*(*Radice*,*Sinistro*,*Destro*) dove il primo argomento del funtore indica la radice dell'albero, il secondo il sottoalbero sinistro e il terzo il sottoalbero destro.

conta(*nil*,0).

conta(*t*(*Radice*,*Sinistro*,*Destro*),*N*):-

conta(*Sinistro*,*N1*),

conta(*Destro*,*N2*),

N is *N1* + *N2* + 1.

Scrivere un predicato Prolog *distinti*(*Tree*, *N*) che restituisca in *N* il numero di informazioni distinte contenute nell'albero.

```
distinti(Tree,N):- visita(Tree,L),
                    conta1(L,N,[]).
visita(nil,[]).
visita(t(Radice,Sinistro,Destro),[Radice|R]):-
    visita(Sinistro,R1),
    visita(Destro,R2),
    append(R1,R2,R).
conta1([],0,-):-!.
conta1([X|L],N,V):-member(X,V),
                    conta1(L,N,V),!.
conta1([X|L],N,V):-conta1(L,N1,[X|V]),
                    N is N1+1,!.

```

Scrivere un programma Prolog che, dato il nodo radice di un albero memorizzato come insieme di fatti $f(P, F)$, dove P é il nodo padre e F il nodo figlio, realizzi la visita dell' albero depth first e ne memorizzi il cammino in una lista.

```
df(Radice,L):-df([Radice],[ ],L).
df([ ],-, [ ]).
df([Nodo|Rest],Visitati,L):-
member(Nodo,Visitati),!,
    df(Rest,Visitati,L).
df([Nodo|Rest],Visitati,[Nodo|L]):-
    figli(Nodo,Figli),
    append(Figli,Rest,Davis),
    df(Davis,[Nodo|Visitati],L).

figli(N,F):-setof(Y,f(N,Y),F).
figli(N,[ ]).
```