

Linguaggi di Programmazione

Daniele De Micheli

2019

Indice

I	Introduzione alla logica	1
1	Logica e Ragionamento	2
1.1	Processo di dimostrazione	3
1.2	Regole di inferenza e calcoli logici	4
2	Programmazione Logica	4
2.1	Stile dichiarativo della programmazione logica	5
2.2	PROLOG	5
2.2.1	Forma Normale Congiuntiva	6
2.2.2	Struttura del Prolog	7
2.2.3	Sintassi del Prolog	7
2.2.4	Modello di esecuzione	10
2.3	L'interprete Prolog: interrogazioni	12
2.3.1	L'interprete prolog: consult	13
2.3.2	Risoluzione ad Input Lineare	13
2.3.3	Alberi di derivazione SLD	15
2.4	Unificazione	16
2.5	Liste in Prolog	17
2.5.1	L'operatore 	17

Parte I

Introduzione alla logica

1 Logica e Ragionamento

Per poter iniziare a parlare di *linguaggi logici*, dobbiamo prima acquisire cosa è un *linguaggio*. Dobbiamo quindi capire come un **ragionamento** può essere **formalizzato** in un numero di **passi** (connessi da **regole**) a partire da **premesse** per raggiungere una **conclusione**.

Questo processo è quello che siamo abituati a riscontrare nella soluzione di *teoremi* tramite **dimostrazioni**.

Un esempio di applicazione di questo processo possiamo vederlo qui di seguito:

Teorema del triangolo isoscele. *Dato un triangolo isoscele, ovvero con due lati $AB = BC$, si dimostra che gli angoli $\angle A$ e $\angle C$ sono uguali.*

Conoscenze pregresse

1. Se due triangoli sono uguali, i due triangoli hanno lati e angoli uguali.
2. Se due triangoli hanno due lati e l'angolo sotteso uguali, allora i due triangoli sono uguali.
3. BH bisettrice di $\angle B$ cioè $\angle ABH = \angle HBC$.

Dimostrazione

- $AB = BC$ per ipotesi;
- $\angle ABH = \angle HBC$ per (3);
- Il triangolo HBC è uguale al triangolo ABH per (2);
- $\angle A$ e $\angle C$ per (1);

Quindi abbiamo trasformato (2) in "**Se** $AB = BC$ e $BH = BH$ e $\angle ABH = \angle HBC$, **allora** il triangolo ABH è uguale al triangolo HBC " e abbiamo trasformato (1) in "**Se** triangolo ABH è uguale al triangolo HBC , **allora** $AB = BC$ e $BH = BH$ e $AH = HC$ e $\angle ABH = \angle HBC$ e $\angle AHB = \angle CHB$ e $\angle A = \angle C$ ".

L'obiettivo diventa a questo punto formalizzare e razionalizzare il processo che permette di affermare

$$AB = BC \vdash \angle A = \angle C$$

dove \vdash indica il simbolo di *derivazione logica*, che comunemente significa "conseguere", "allora", ecc.

Formalizzazione

Abbiamo assunto che:

- $\mathbf{P} = \{AB = BC, \angle ABH = \angle HBC, BH = HB\}$.

Avevamo inoltre delle conoscenze pregresse (vedi *conoscenze pregresse* sopra riportate). Abbiamo quindi costruito una catena di **formule**:

Tabella 1: Triangolo Isoscele.

Formule	Origine
P1: $AB = BC$	da P 10.1
P2: $\angle ABH = \angle HBC$	da P
P3: $BH = HB$	da P
P4: $AB = BC \wedge BH = HB \wedge \angle ABH = \angle HBC$	da P1, P2, P3 e introduzione della congiunzione
P5: $\triangle ABH = \triangle HBC$	da P4 , regola ₁ e Modus Ponens
P6: $AB = BC \wedge BH = BH \wedge AH = HC$ $\wedge \angle ABH = \angle HBC \wedge \angle AHB = \angle CHB \wedge$ $\angle A = \angle C$	da P5 , regola ₂ e Modus Ponens
P7: $\angle A = \angle C$	da P6 e l'eliminazione della congiunzione (il simbolo \wedge)

Le parti evidenziate in rosso nella tabella 1 sono le *regole di inferenza*.

1.1 Processo di dimostrazione

Una "prova" D , dove **S** è l'insieme delle "affermazioni note" e F la frase (es. la formula) che vogliamo provare.

$$D \equiv \mathbf{S} \vdash F$$

(che si legge: F è una conseguenza di **S**) è una sequenza di passi

$$D = \langle P_1, P_2, \dots, P_n \rangle$$

dove

$$P_n = F$$

$$P_i \in S \cup \{P_j \mid j < i\}$$

o P_i può essere ottenuto da P_{i1}, \dots, P_{im} (con $i1 < i, \dots, im < i$) mediante l'applicazione di una regola di inferenza.

1.2 Regole di inferenza e calcoli logici

Un insieme di regole di inferenza costituisce la base di un calcolo logico. Diversi insieme di regole danno vita a diversi calcoli logici. Lo scopo di un calcolo logico è quello di manipolare delle formule logiche in modo completamente **sintattico** al fine di stabilire una connessione tra un insieme di formule di *partenza* (di solito un insieme di formule dette *assiomi*) ed un insieme di *conclusioni*.

2 Programmazione Logica

La programmazione logica nasce all'inizio degli anni settanta da studi sulla deduzione automatica: il **Prolog** costituisce uno dei suoi risultati principali. Essa non è soltanto rappresentata dal Prolog; costituisce infatti un settore molto ricco che cerca di utilizzare la logica matematica come base dei linguaggi di programmazione.

Gli obiettivi del linguaggio di programmazione logica sono:

- semplicità del formalismo;
- linguaggio ad alto livello;
- semantica chiara;

Questo tipo di linguaggio si focalizza sulle solide basi della *logica matematica*. Con l'avvento dell'informatica difatti si è sempre più utilizzata la logica matematica per dimostrare teoremi tramite i calcolatori (che permettono di ottenere risultati in minor tempo e con meno errori). Tra le procedure utilizzate si ricordano la *procedura di Davis e Putnam* e il *principio di risoluzione*.

Per rendere bene l'idea, la programmazione logica viene utilizzata per verificare la correttezza di altri software, per rappresentare la conoscenza di Intelligenza Artificiale o ancora per il formalismo nei database (come Datalog).

2.1 Stile dichiarativo della programmazione logica

Lo stile della programmazione logica ha delle precise caratteristiche:

- Un programma è un *insieme di formule*.
- Possiede un grande potere espressivo.
- Il processo di risoluzione prevede la costruzione di una dimostrazione logica di un'affermazione (**goal**).
- Possiede una **base formale**:
 - Calcolo dei predicati del primo ordine (vedi sez. ????) ma con limitazione nel tipo di formule (**clausole di Horn** (1))
 - Utilizzo di particolari tecniche per la dimostrazione di teoremi (meccanismo di **Risoluzione**)

2.2 PROLOG

Il Prolog (acronimo di **PRO**gramming in **LOGic**) fu ideato e realizzato nel 1973 da Robert Kowalski (aspetto teorico) e Marten Van Emdem (dimostrazione sperimentale). Esso si basa su una restrizione della *logica del primo ordine*. Come caratteristica base dei linguaggi logici, anche Prolog utilizza uno stile dichiarativo di programmazione. La sua primaria funzione è quella di determinare se una certa affermazione è vera oppure no e, se è vera, quali vincoli sui valori attribuiti alle variabili hanno generato la risposta.

Formule ben formate Le formule ben formate (*fbf*, o **well-formed formula**, *wff*) di un linguaggio logico del primo ordine può essere riscritta in **forma normale a clausole**.

Vi sono due forme normali a clausole:

- **Forma normale congiunta** (conjunctive normal form - CNF): la formula è una **congiunzione** di **disgiunzioni** di predicati o di negazioni di predicati (letterali *positivi* o letterali *negativi*).

$$\bigwedge_i \left(\bigvee_j L_{ji} \right) \quad (1)$$

- **Forma normale disgiunta** (disjunctive normal form - DNF): la formula è una **disgiunzione** di **congiunzioni** di predicati o di negazione di predicati (letterali *positivi* o letterali *negativi*).

$$\bigvee_j \left(\bigwedge_i L_{ji} \right) \quad (2)$$

dove

$$L_{ij} \equiv P_{ij}(x, y, \dots, z) \circ L_{ij} \equiv \neg Q_{ij}(x, y, \dots, z)$$

2.2.1 Forma Normale Congiuntiva

Consideriamo una *wff* in CNF (1). Per esempio:

$$\underbrace{(p(x) \vee q(x, y) \vee \neg t(z))}_{\text{clausola1}} \wedge \underbrace{(p(w) \vee \neg s(u) \vee \neg r(v))}_{\text{clausola2}} \quad (3)$$

se scartiamo il simbolo di **congiunzione** (3), rimaniamo con solo le clausole disgiuntive

1. $(p(x) \vee q(x, y) \vee \neg t(z))$
2. $(p(w) \vee \neg s(u) \vee \neg r(v))$

Le clausole così ottenute sono anche riscrivibili come

1. $t(z) \Rightarrow p(x) \vee q(x, y)$
2. $s(u) \wedge r(v) \Rightarrow p(w)$

ovvero un insieme di formule in CNF è riscrivibile come un insieme (congiunzione) di implicazioni.

Clausole di Horn 1. *Le clausole che hanno al più un solo letterale positivo (con o senza letterali negativi) prendono il nome di **Clausole di Horn**.*

Detto questo, abbiamo che:

- Non tutte le fbf possono essere trasformate in un insieme di clausole di Horn.
- *I programmi Prolog sono collezioni di clausole di Horn*

2.2.2 Struttura del Prolog

Il linguaggio Prolog:

- Non contiene (quasi) istruzioni
- Contiene solo fatti e regole (clausole di Horn):
 - **Fatti**: asserzioni vere nel contesto che stiamo descrivendo.
 - **Regole**: ci danno gli strumenti per dedurre nuovi fatti da quelli esistenti.
- Un programma Prolog ci dà informazioni su un sistema ed è normalmente chiamato **base di conoscenza** (knowledge base)
- Un programma Prolog non si "esegue" ma si "*interroga*" (**queried**): al programma si chiede se i fatti sono veri; esso risponderà con un **True** o **False** alla domanda.

2.2.3 Sintassi del Prolog

Un programma Prolog è costituito da un insieme di clausole della forma

Tabella 2: Sintassi Prolog

$a.$	FATTO / ASERZIONE
$c :- b_1, b_2, \dots, b_n.$	REGOLA
$:- q_1, q_2, \dots, q_m$	GOAL
$? - q_1, q_2, \dots, q_m$	QUERY

In cui a, b_i e q_i sono **termini** (compositi). Da notare che in molte implementazioni il prompt Prolog è un operatore che chiede al sistema di valutare il **goal**, in questo caso una congiunzione di termini.

Termini Le espressioni nel Prolog sono chiamate **termini**. Esistono diversi tipi di termini:

- ATOMI :

- Una sequenza di caratteri alfanumerici, che inizia con un carattere minuscolo (può contenere il carattere " _ underscore)
- Qualsiasi cosa racchiusa tra apici singoli (' ')
- Un numero
- Una stringa (SWI Prolog)

◦ **VARIABILI:**

- Una variabile (logica) è una sequenza alfanumerica che inizia con un carattere **MAIUSCOLO** o con il carattere _ (underscore)
- Le variabili (notare il plurale) composte solo dal simbolo _ sono chiamate **indifferenza** (*don't care o any*) o **anonime**
- Le variabili vengono **istanziate** (legate a un valore) con il procedere del programma (nella dimostrazione del teorema)

◦ Una composizione di termini \Rightarrow **TERMINE COMPOSTO** (simbolo di *fun-*
tore più uno o due argomenti):

- + Un **funtore** (simbolo di funzione o di predicato definito come atomo)
- + Una sequenza di termini racchiusi tra parentesi tonde e separati da virgole. Questi sono chiamati argomenti del funtore



Non ci deve mai essere uno spazio tra il funtore e la parentesi di sinistra; questo per via di caratteristiche molto sofisticate del sistema di parsing di Prolog (cfr., operatori)



Prolog: fatti e predicati Un **fatto** (**predicato**) consiste in:

- Un nome di predicato, ad esempio *fattoriale*, *genitore*, *uomo* o *animale*; deve iniziare con una lettera **minuscola**.
- Zero o più argomenti come *maria*, *42* o *cane*. Da tenere presente che i fatti (e le regole e le domande) **devono** essere terminati da un punto (".").

Le regole In Prolog si usano le **regole** quando si vuole esprimere che un certo fatto dipende da un insieme di altri fatti. Per esprimere in linguaggio naturale questa dipendenza usiamo la parola "**se**" ("if" in inglese).

Ad esempio:

- *Uso l'ombrello **se** piove;*
- *Luca mangia la pizza **se** non contiene glutine;*

Le regole sono anche usate per esprimere definizioni. Ad esempio:

X è un pesce se:

- X è un animale
- X ha le squame

Una regola è costituita da una **testa** e da un **corpo**. Sono inoltre caratterizzati come segue:

1. Testa e corpo sono collegati dall'operatore ":-".
2. La testa di una regola corrisponde al **conseguente** di un'implicazione logica.
3. Il corpo di una regola corrisponde all'**antecedente** di un'implicazione logica.
4. Le regole Prolog corrispondono alle **clausole di Horn**, ovvero hanno un solo termine (predicato) come conseguente.
 - L'operatore Prolog ":-" esprime il "**se**" (*implicazione*).
 - L'operatore Prolog "," equivale a "**e**" (**and**, o *congiunzione*).

Un esempio di regola Prolog può essere:

"un pesce è un animale che ha le squame" diventa:

pesce(X) :- animale(X), ha_le_squame(X).

Una relazione può essere però definita per necessità da due regole (o clausole) aventi lo stesso predicato come conclusione. Per esempio:

genitore(X, Y) :- padre(x, Y).

genitore(X, Y) :- madre(x, Y).

Le regole (ed i fatti) sono implicitamente connesse dall'operatore logico di congiunzione ("**and**"); se non si sono commessi errori logici, **entrambe** le implicazioni soprastanti sono da ritenersi **vere**.

Una relazione può essere definita anche *ricorsivamente*. In questo caso la definizione richiede almeno due proposizioni: una è quella ricorsiva che corrisponde al caso generale, l'altra esprime il caso base più semplice.

$antenato(X, Y) : - genitore(X, Y).$

$antenato(X, Y) : - genitore(Z, Y), antenato(X, Z).$

Operatori Logici : gli operatori logici "and" e "or" possono essere utilizzati nelle regole tramite una specifica sintassi. L'operatore "**AND**" viene inserito in una regola con il carattere *virgola* ','; L'operatore "**OR**" viene inserito in una regola con il carattere *punto e virgola* ';.

RICAPITOLANDO Bisogna ricordare che:

1. ogni fatto o regola DEVE terminare con un punto '.'
2. ogni variabile DEVE iniziare con una MAIUSCOLA
3. i commenti si inseriscono dopo un "%" (commento in linea) o tra '/' e '*' (come in altri linguaggi).

2.2.4 Modello di esecuzione

Come abbiamo già visto, il Prolog utilizza un'interpretazione dichiarativa. Per esempio,

$$p : - q, r. \quad (4)$$

si può leggere come "*p è vero se sono veri q e r*". Abbiamo anche una interpretazione procedurale, in cui il problema *p* viene scomposto nei sottoproblemi *q* e *r*. In questa visione, un *goal* può essere visto come una chiamata ad una procedura in cui la testa è l'intestazione mentre la parte destra è il corpo.

Controllo di esecuzione Come abbiamo abbiamo intuito, le clausole nel data base di un programma Prolog vengono considerate "da sinistra, verso destra" e "dall'alto al basso". Se un (sotto)goal fallisce, allora il dimostratore Prolog sceglie un'alternativa, scandendo "dall'alto verso il basso" la lista delle clausole (nella pratica, esegue in ordine le istruzioni che trova).

Per poter manipolare l'esecuzione, il Prolog dispone di un *predicato* speciale, chiamato **cut** (*taglio*, scritto con il solo simbolo esclamativo "!"). Le principali caratteristiche del **cut** sono le seguenti:

- Il cut è molto complesso da interpretare (non ha un'interpretazione logica ma bensì procedurale);
- La sua importanza non può essere sottovalutata; è un predicato potente, difficile da usare nel modo corretto e particolarmente utile se usato correttamente.
- Per capire come funziona è necessario avere un'idea del funzionamento del dimostratore Prolog.

Il predicato cut '!' Consideriamo la seguente clausola generica con **cut:**

$$C = a : -b_1, b_2, \dots, b_k, !, b_{k+1}, \dots, b_n$$

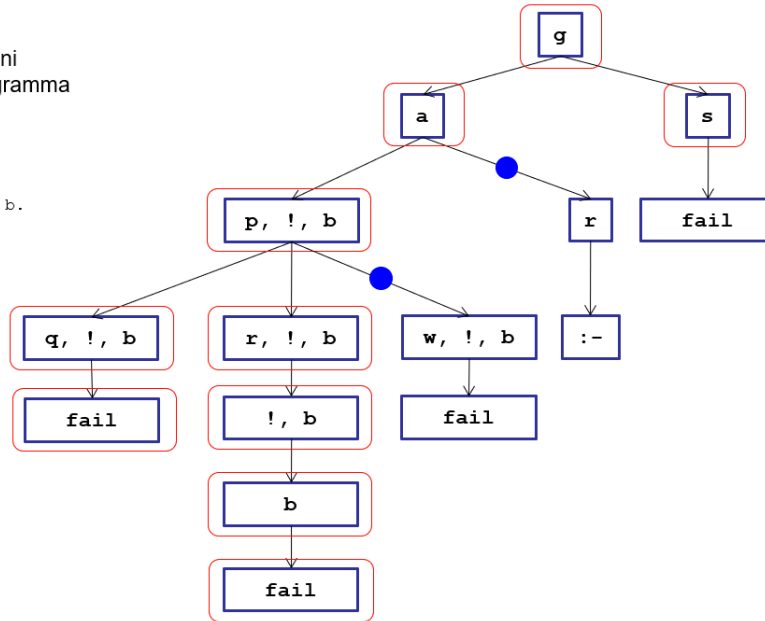
L'effetto del *cut* è il seguente:

- Se il goal corrente unifica con a e b_1, \dots, b_k hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G .
- Ogni clausola alternativa (successiva, in basso) per a che unifica con G viene ignorata.
- Se un qualche b_j con $j > k$ fallisse, il backtracking si fermerebbe al *cut*. Le altre scelte per b_i con $i \leq k$ sono di conseguenza rimosse dall'albero di derivazione.
- Quando il backtracking raggiunge il *cut*, allora il *cut* fallisce e la ricerca procede dall'ultimo punto di scelta prima che G scegliesse C .

Qui di seguito è proposto un esempio di goal e di clausole contenente il *cut* e di cui viene mostrato l'albero di derivazione.

Albero di derivazioni
SLD/LM per il programma
con goal :- g.

(cl1) g :- a.
(cl2) g :- s.
(cl3) a :- p, !, b.
(cl4) a :- r.
(cl5) p :- q.
(cl6) p :- r.
(cl7) p :- w.
(cl8) r.



Nella figura si può notare il percorso di dimostrazione che l'interprete Prolog segue. Si può anche notare che tra i percorsi "tagliati" fuori dal risultato (quelli posti dopo il pallino blu nel collegamento tra padre e figlio) c'è pure l'unico **successo** possibile della query rispetto alle clausole di programma. In questo caso infatti possiamo dire che il *cut* ha escluso la soluzione dai risultati.

Possiamo anche distinguere tra due tipi di *cut*, o meglio, tra due utilizzi del *cut*:

- **Green Cuts:** utili per esprimere "determinismo" (e quindi per rendere più efficiente il programma).
- **Red Cuts:** usati per soli scopi di efficienza, hanno come caratteristica principale quella di omettere alcune condizioni esplicite in un programma e, *soprattutto*, quella di modificare la semantica del programma equivalente senza *cuts*. Questo utilizzo è considerato generalmente sbagliato, anche se in alcuni casi può rivelarsi utile.

2.3 L'interprete Prolog: interrogazioni

Una volta che le regole e i fatti sono stati "caricati" nell'interprete, eseguire un programma Prolog equivale a *interrogare* l'**interprete**. Una volta fatto partire,

l'interprete Prolog ci presenta un prompt così:

?–

Come si può intuire anche dalla sintassi del prompt, interrogare l'interprete non è altro che porre una *domanda* a cui generalmente il Prolog risponde "**true**" o "**false**". Le interrogazioni possono contenere variabili interpretate come *variabili esistenziali*. Queste sono istanziate quando il Prolog prova a rispondere alla domanda fattagli. Tutte le variabili istanziate vengono mostrate nella risposta insieme a "true".

2.3.1 L'interprete prolog: consult

Nel Prolog la base di conoscenza è *nascosta* ed è accessibile solo tramite opportuni comandi o tramite ambiente di programmazione. Ovviamente è necessario poter *inizializzare* o *caricare* un insieme di fatti e regole nell'ambiente Prolog. Il comando principale che assolve questa funzione è **consult**:

- Il comando consult appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento.
- Il file deve contenere un insieme di fatti e regole.

2.3.2 Risoluzione ad Input Lineare

Il sistema Prolog dimostra la veridicità o meno di un'interrogazione (un goal) eseguendo *una sequenza di passi di risoluzione*. L'ordine complessivo in cui questi passi vengono eseguiti rende i sistemi di prova di teoremi basati su risoluzione più o meno "*efficaci*".

Infatti, in Prolog la risoluzione avviene sempre tra l'ultimo goal derivato in ciascun passo e una "clausola di programma"; non avviene mai tra due clausole di programma o fra una clausola di programma ed un goal derivato in precedenza. Questa particolare forma di risoluzione prende il nome di **Risoluzione SLD** (*Selection function for Linear and Definite sentences Resolution*) in cui le frasi "lineari" sono essenzialmente le clausole di Horn.

Questo metodo di risoluzione funziona in modo semplice ed efficace. Prendiamo ad esempio il seguente goal G_j :

$$G_j \equiv ? - A_{i,1}, A_{i,2}, A_{i,3}, \dots, A_{i,m}. \quad (5)$$

e la regola

$$A_r : - B_{r,1}, B_{r,2}, \dots, B_{r,k}. \quad (6)$$

se esiste un unificatore σ tale che $\sigma[A_r] = \sigma[A_{i,1}]$, allora si ottiene un nuovo goal G_{j+1}

$$G_{j+1} \equiv ? - B_{r,1}^1, B_{r,2}^1, \dots, B_{r,k}^1, A_{i,2}^1, A_{i,3}^1, \dots, A_{i,m}^1. \quad (7)$$

Questo è **un** passo di risoluzione eseguito dal sistema Prolog (in cui A^1 e B^1 sono i risultati $\sigma[A] = A^1$ e $\sigma[B] = B^1$. In altre parole, potremmo scrivere l'esempio come sequenza di passaggi:

$$(P1) \quad A_r (= A_{i,1}) : -\neg B_1 \wedge B_2 \wedge \dots \wedge B_n$$

$$(P2) \quad \neg A_{i,1} \vee \neg A_{i,2} \vee \dots \vee \neg A_{i,m}$$

$$(P3) \quad A_{1,1}, \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n$$

$$(P4) \quad \neg B_1 \vee \neg B_2 \dots \vee \neg B_n \vee \neg A_{i,2} \vee \neg A_{i,3} \dots \vee \neg A_{i,m}$$

Noi sappiamo che $A_{i,1}$ può essere risolto con A_r . Ma A_r è a sua volta una query (P1). Il primo termine della query può essere unificato con il negato della seconda, il quale produce il nuovo goal (P4).

In una soluzione SLD, il passo di risoluzione avviene sempre tra l'ultimo goal e una clausola di programma. Il risultato può essere

- **SUCCESS**: viene generata la clausola vuota, ovvero se per finito, G_n è uguale alla clausola vuota $G_n \equiv -$
- **FINITE INSUCCESS**: se per n finito, G_n non è uguale alla clausola vuota e non è più possibile derivare un nuovo *risolvente* da G_n ed una clausola di programma.
- **INFINITE INSUCCESS**: se è possibile derivare nuovi *risolventi* tutti diversi dalla clausola.

Durante il processo di generazione di goal intermedi si costruiscono delle variabili dei letterali e delle clausole coinvolti mediante la rinominazione di variabili. La rinominazione avviene nel momento in cui si genera un nuovo goal ma ci si deve portare dietro le variabili che non sono state unificate, per esempio.

Un esempio di **renaming** di variabili è il seguente: presa

$$p(X) : -q(X, g(z)).$$

è equivalente alla clausola

$$p(X1) : -q(X1, g(FooFrobboz)).$$

rinominata.

Per applicare la risoluzione del goal corrente ci possono essere diverse clausole di programma adeguate al compito. Ci sono diverse strategie, di cui due sono le seguenti:

1. **In profondità (*Depth First*):** si sceglie una clausola e si mantiene fissa questa scelta, finché non si arriva alla clausola vuota o alla impossibilità di fare nuove risoluzioni; in questo ultimo caso si riconsiderano le scelte fatte precedentemente.
2. **In ampiezza (*Breadth First*):** si considerano in parallelo tutte le possibili alternative.

Il Prolog adotta una strategia di risoluzione "*in profondità*" con [backtracking](#). Questo permette di risparmiare memoria ma non è completa per le clausole di Horn.

2.3.3 Alberi di derivazione SLD

Dato un programma logico P , un goal G_0 e una regola di calcolo R , un **albero SLD** per $P \cup \{G_0\}$ via R è definito sulla base del processo di prova visto precedentemente;

- Ciascun nodo dell'albero è un goal (possibilmente vuoto)
- La radice dell'albero SLD è il goal G_0
- Dato il nodo

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

se A_m è il sottogoal *selezionato* dalla regola di calcolo R , allora questo nodo (genitore) ha un nodo figlio per ciascuna clausola del tipo

$$C_i \equiv A_i : -B_{i,1}, \dots, B_{i,q}$$

$$C_k \equiv A_k.$$

di P tale che A_i e A_m (A_k e A_m) sono unificabili attraverso la sostituzione più generale σ .

- Il nodo figlio è etichettato con la clausola goal

$$: -\sigma[A_1, \dots, A_{m-1}, B_{i,1}, \dots, B_{i,q}, A_{m+1}, \dots, A_k]$$

$$: -\sigma[A_1, \dots, A_{m-1}, A_{m+1}, \dots, A_k]$$

e il ramo dal nodo padre al figlio è etichettato dalla sostituzione σ e dalla clausola delezionata C_i (o C_k).

- Il nodo vuoto (indicato con ":-") non ha figli.

2.4 Unificazione

L'operazione di istanziazione di variabili durante la "prova" di un predicato è il risultato di una procedura particolare, detta unificazione. Dati due termini, la procedura di unificazione crea un insieme di *sostituzioni* delle variabili, il quale permette di rendere "**uguali**" i due termini.

Tradizionalmente la procedura di unificazione costruisce un insieme di sostituzioni chiamato "**most general unifier**" ed indicato con *Mgu*. Una sostituzione è indicata come una sequenza (ordinata) di coppie variabile/valore.

Esempi:

$Mgu(42, 42)$	$\Rightarrow \{\}$
$Mgu(42, X)$	$\Rightarrow \{X/42\}$
$Mgu(X, 42)$	$\Rightarrow \{X/42\}$
$Mgu(foo(bar, 42), foo(bar, X))$	$\Rightarrow \{X/42\}$
$Mgu(foo(Y, 42), foo(bar, X))$	$\Rightarrow \{X/42\}$
$Mgu(foo(X), foo(bar(Y)))$	$\Rightarrow \{X/bar(Y), Y/_G001\}$

Notare l'ultimo esempio con **ridenominazione** di variabili ($_G001$ è il nome assegnato alla variabile che contiene il risultato. Esso è generato dal programma Prolog). Il "*most general unifier*" non è nient'altro che il risultato finale della procedura di valutazione - ovvero di prova - del Prolog.

Il modo più semplice per vedere come la procedura di unificazione funziona è di usare l'operatore Prolog = (detto, per l'appunto, **operatore di unificazione**).

Esempi:

? – **42 = 42.**

Yes

? – **42 = X.**

$X = 42$

Yes

? – **foo(Y, 42)=foo(bar, X).**

$Y = bar$

$X = 42$

Yes

2.5 Liste in Prolog

Si definisce una *lista* in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre '[' e ']' e separandoli da virgole. Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste. Per indicare una lista vuota si usa la notazione '[]'.

Ogni lista non vuota può essere divisa in due parti, una *testa* ed *coda*:

- La *testa* è il primo elemento della lista.
- La *coda* rappresenta tutto il resto della lista ed è **SEMPRE** una lista.

2.5.1 L'operatore |

Prolog possiede uno speciale operatore usato per distinguere tra l'inizio e la coda di una lista: l'operatore "|".

Esempi dell'operatore | :

? – [X | Ys] = [mia, vincente, jules, yolanda].

X = mia

Ys = [vincent, jules, yolanda]

Yes

? – [X, Y | Zs] = [the, answer, is, 42].

X = the

Y = answer

Zs = [is, 42]

Yes

? – [X, 42 | _] = [41, 42, 43, foo(bar)].

X = 41

Yes