Strategia **Linear Input**: (non completa) sceglie sempre una clausola “parent” nell’insieme base (il programma) mentre la seconda clausola “parent” è il risolvente derivato al passo precedente (il goal)

Una **variante** per una clausola C è la clausola C′ ottenuta da C rinominando le sue variabili (renaming)

* p(X) :- q(X, g(Z)).
* p(X1) :- q(X1, g(Z1)).

Un **programma PROLOG** è un **insieme di clausole di Horn** che **rappresentano**:

* **Fatti** riguardanti gli oggetti in esame e le relazioni che intercorrono
* **Regole** sugli oggetti e sulle relazioni (se . . . allora)
* **Goal** (clausole senza testa), negazione della query sulla base della conoscenza definita

Le proprietà di **correttezza e completezza** della risoluzione SLD per le clausole Horn **non valgono** per Prolog. Prolog **non utilizza l’occur check** nell’algoritmo di unificazione (si

perde la correttezza, deriva formule che non seguono logicamente). **Usa la strategia di ricerca depth-first**, non completa, per esplorare lo spazio delle soluzioni rappresentato dall’albero SLD (perde la completezza), non tutte le conseguenze logiche sono dimostrabili

**Unificazione**:

|  | C2 | X2 | S2 |
| --- | --- | --- | --- |
| C1 | If C1 == C2 | X2/C1 | No |
| X1 | X1/C2 | X1/X2 | X1/S2 |
| S1 | No | X2/S1 | If =funtore, e parametri unificabili |

In **Prolog**, si usa la regola che **seleziona l’atomo più a sinistra nel goal corrente**, si **applicano le clausole secondo il loro ordine testuale** e nell’algoritmo di unificazione non si esegue l’Occur Check (es. X = f(X) unifica in Prolog)

**Derivazione SLD**

Tre tipi di derivazione:

* **Successo** (detta anche refutazione) se per n finito Gn è uguale alla clausola vuota.
* **Fallimento finito**: se per n finito non è più possibile derivare un nuovo risolvente.
* **Fallimento infinito**: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.

Una **risposta calcolata** q per P ∪ {G0} è la **sostituzione ottenuta restringendo la composizione delle sostituzioni MGU** q1, …, qn utilizzate nella refutazione SLD di P ∪ {G0} alle variabili di G0.

La risposta calcolata o **sostituzione di risposta calcolata è il “testimone” del fatto che esiste una dimostrazione costruttiva di una formula** quantificata esistenzialmente (G0).

**Albero SLD**

É lo **spazio delle derivazioni SLD per un goal G0 dall’insieme di clausole definite P**. È un albero OR. Raccoglie i tre tipi di derivazioni: Successo, Fallimento finito e Fallimento infinito.

Nella **risoluzione SLD** così come è stata enunciata si hanno **due forme di non determinismo**:

* **Selezione di un atomo** Am del goal da unificare con la testa di una clausola, e viene risolta definendo una particolare regola di calcolo (o funzione di selezione). Prolog usa la **regola left-most**, essa **non ha impatto sulla completezza**.
* **Scelta di quale clausola** del programma P **utilizzare** in un passo di risoluzione, e viene risolta adottando una strategia di ricerca, nello spazio delle soluzioni che è l’albero SLD. Prolog usa la **strategia depth-first**, questa strategia (non completa) **ha impatto sulla completezza**.

Il linguaggio Prolog, **adotta la strategia in profondità con backtracking** perché può essere realizzata in modo efficiente attraverso un unico stack di goal. Tale stack rappresenta il ramo che si sta esplorando e contiene opportuni riferimenti ai rami alternativi da esplorare in caso di fallimento (punti di scelta, o choice point). Per quello che riguarda la scelta fra nodi fratelli, la strategia Prolog li ordina seguendo l’ordine testuale delle clausole che li hanno generati.

Aritmetica e Operazioni Relazionali

In questo modo ogni espressione può essere rappresentata come un termine Prolog

* Operatori unari: - exp log ln sin cos tan
* Operatori binari: + - \* \ div mod

Data un’espressione, è necessario un meccanismo per la valutazione- Speciale predicato predefinito **is**

* T is Expr: is(T,Expr)
  + T può essere un atomo numerico o una variabile
  + Expr deve essere un’espressione
  + L’espressione Expr viene valutata e il risultato della valutazione viene unificato con T
  + Le variabili in Expr devono essere istanziate al momento della valutazione

Nel caso dell’operatore **is l’ordine dei goal è rilevante**.

Il goal: X is 2+3, Y is X+1 ha successo e produce la coppia di istanziazioni X=5, Y=6. Mentre il goal: Y is X+1, X is 2+3 fallisce.

Il Prolog fornisce operatori relazionali per confrontare i valori di espressioni. Tali operatori sono utilizzabili come goal all’interno di una clausola Prolog ed hanno notazione infissa

* Operatori Relazionali: > < >= =< =:= =/=

Esempi:

* T1 = T2: **Verifica se T1 e T2 sono unificabili**. Viene generata la sostituzione (MGU) che unifica i due termini (e vengono pertanto legate le variabili nei due termini).
* T1 == T2: **Verifica se T1 e T2 sono uguali (identici)**. In particolare, se i due termini contengono due variabili in posizioni equivalenti, perché i termini siano uguali, le due variabili devono essere la stessa variabile. Si noti che in questo caso non viene generata alcuna sostituzione. **Non li valuta anche se rappresentano espressioni**.
* Expr1 =:= Expr2: **Verifica se Expr1 e Expr2 sono espressioni che hanno lo stesso valore**.
* T1 \= T2: **Verifica se T1 e T2 non sono unificabili** 2+3 \= 5. true
* T1 \== T2: **Verifica se T1 e T2 non sono uguali (identici)**; ha successo se i due termini (non valutati) non sono identici: a \== b. true
* Expr1 =\= Expr2: **Ha successo se le due espressioni (che vengono valutate) non hanno lo stesso valore**.

Il Prolog non fornisce alcun costrutto sintattico per l’iterazione (quali, ad esempio, i costrutti while e repeat) e **l’unico meccanismo per ottenere iterazione è la definizione ricorsiva**. In programmazione (anche imperativa) una funzione f è definita per **ricorsione tail** **se f è la funzione “più esterna” nella definizione ricorsiva o, in altri termini, se sul risultato della chiamata ricorsiva di f non vengono effettuate** ulteriori operazioni.

In Prolog un predicato viene individuato univocamente dal nome e dal numero degli argomenti (arietà)

I principali predicati di I/O in Prolog sono:

* read(X) Legge un termine da console
* write(X) Stampa X (il termine a cui è legata X) a video
* nl Stampa a video un “a capo” (newline)

Liste

La definizione di lista può essere data ricorsivamente nel modo seguente:

* Con: [] si rappresenta la lista vuota
* Il termine [T | Lista] è una lista, dove **T** (un termine qualsiasi) **è la Testa e Lista** (una lista a sua volta) **la Coda o Resto**.

Esempi:

* []
* [a | []] → [a]
* [a | [b | []]] → [a,b]
* [f(g(X)) | [h(z) | [c | []]]] → [f(g(X)), h(Z), c]
* [[] | []] → [[]]
* [[a | []] | [b | []]] → [[a], b]

Esempi delle operazioni sulle liste:

* length([b, a, b, [c,a], d], X).
  + true, x = 5
* member(X, [[1,2,3], [a], [d,c]]).
  + true, x = [1,2,3], x = [a], x = [d,c]
* member(1, [[1,2,3], [a], [d,c]]).
  + false

Operazioni:

| Verifica se L è una lista | is\_list([]).  is\_list([\_|L]) :- is\_list(L). |
| --- | --- |
| Verifica se T è nella lista | member(T, [T|\_]).  member(T, [\_|L]) :- member(T, L). |
|  | append([], L, L).  append([H|T], L2, [H|T1]) :- append(T, L2, T1). |
|  | reverse([], []).  reverse([H|T], R):- reverse(T, T2),  append(T2, [H], R). |
|  | length([], 0).  length([\_|L], N) :- length(L, N1),  N is N1 + 1. |
|  | sum\_list([], 0).  sum\_list([H|T], R) :- sum\_list(T, R1),  R is R1+H. |
|  | depth\_member(T, [T|\_]).  depth\_member(T, [L|\_]) :- is\_list(L),  depth\_member(T, L).  depth\_member(T, [\_|L]) :- depth\_member(T, L). |
|  | delete1(E, [], []).  delete1(E, [E|T], T).  delete1(E, [H|T], [H|T1]) :- T\==H  delete1(E, T, T1). |
|  | delete(E, [], []).  delete(E, [E|T], T1) :- delete(E, T, T1).  delete(E, [H|T], [H|T1]) :- T\==H  delete(E, T, T1). |

Controllo del Backtracking

Predicati predefiniti che consentono di influenzare e controllare il processo di esecuzione (dimostrazione) di un goal

* **Predicato cut !**
  + è denotato dal simbolo !
  + è un predicato extra-logico (può risultare difficile da comprendere ...)
  + Da usare con attenzione, quando siamo sicuri di non togliere soluzioni

*Per ogni valutazione della query gol, vi è uno stack di esecuzione in vi è l’atomo del gol, però, si genera un punto di scelta con le possibilità che unificano con quell’atomo (altro stack che tiene traccia di questi). Con un fallimento, svuoto lo stack fino al punto di scelta più vicino (in ordine cronologico). Quando scelgo l’altra possibilità, si elimina il punto di scelta*.

1. **Stack di esecuzione** che contiene i record di attivazione (environment) delle varie procedure
2. **Stack di backtracking** che contiene l’insieme dei punti di scelta (choice point). Ad ogni fase della valutazione tale stack contiene puntatori alle scelte aperte nelle fasi precedenti della dimostrazione.

In realtà è utilizzato un solo stack, con alternanza di environment e choice point; le linee rosse indicate rappresentano i choice point.

**Cut**: L’effetto è quello di rendere definitive alcune scelte fatte nel corso della valutazione dall’interprete Prolog ossia **quello di eliminare alcuni choice point dallo stack di backtracking**.

Data la clausola: p :- q1, q2 …, qi, !, qi+1, qi+2, …, qn.

* **Tutte le scelte fatte nella valutazione dei goal q1, q2, …, qi** **e** in quella del goal **p vengono rese definitive**; in altri termini, tutti i punti di scelta per tali goal (per le istanze di tali goal utilizzate) vengono rimossi dallo stack di backtracking.

| Member più efficiente: verifica se T è nella lista una sola volta | member(T, [T|\_]) :- !.  member(T, [\_|L]) :- member(T, L). |
| --- | --- |

Per questa versione di member, non possiamo però:

* Individuare tutti gli elementi di una lista
* Verificare l’appartenenza multipla di un elemento alla lista

| Mutua esclusione  Senza il cut, si eseguirebbe in backtracking anche la seconda | p(X) :- a(X), !, b.  p(X) :- c. |
| --- | --- |
| Cancellazione del primo elemento unificabile con E | delete1(E, [], []) :- !  delete1(E, [E|T], T) :- !  delete1(E, [H|T], [H|T1]) :- delete1(E, T, T1) |
| Cancellazione di tutti gli elementi unificabili con E | delete(E, [], []) :- !  delete(E, [E|T], T1) :- !, delete(E, T, T1)  delete(E, [H|T], [H|T1]) :- delete(E, T, T1) |
| Merge di due liste ordinate | merge([], L2, L2) :- !.  merge(L1, [], L1) :- !.  merge([X|REST1], [X|REST2], [X,X|REST]) :- !,  merge(REST1, REST2, REST).  merge([X|REST1], [Y|REST2], [X|REST]) :-  X < Y, !, merge(REST1, [Y|REST2], REST).  merge([X|REST1], [Y| REST2], [Y|REST]) :-  merge([X|REST1], REST2, REST). |

Trattare informazioni negative

Questo è l’approccio seguito nelle basi di dati in cui, per problemi di dimensioni, si memorizzano solo le informazioni positive. Questo significato intuitivo è espresso dalla regola di inferenza nota come **Closed World Assumption** (CWA) o Ipotesi di Mondo Chiuso

* Se un atomo “ground” A non è conseguenza logica di un programma P, allora si può inferire ∼A.

Esempi:

* capitale(roma). citta(X) :- capitale(X). citta(bologna).
  + Con la CWA è possibile inferire ~capitale(bologna)
* capitale(roma). citta(X) :- capitale(X). citta(bologna). capitale(bologna).
  + Con la CWA non è più possibile inferire ~capitale(bologna)

L’aggiunta di nuovi assiomi alla teoria (ossia nuove clausole in un programma) ha modificato l’insieme di teoremi che valevano precedentemente.

A causa dell’indecidibilità (semi-decidibilità) della logica del primo ordine, non esiste alcun algoritmo in grado di stabilire in un tempo finito se A non è conseguenza logica di P

* Dal punto di vista operazionale, se A non è conseguenza logica di P, la risoluzione SLD può non terminare.
* Esempio:
  + citta(roma) :- citta(roma). citta(bologna).
    - citta(roma) non è conseguenza logica di P e
      * CWA(P) = {∼citta(roma)} ma la risoluzione SLD per citta(roma) non termina

L’applicazione della **CWA deve necessariamente essere ristretta agli atomi la cui dimostrazione termina in tempo finito**, cioè agli atomi per cui la risoluzione SLD non diverge.

**Negazione per Fallimento** (Negation as Failure – NF) [Clark 1978] si limita a **derivare la negazione di atomi la cui dimostrazione termina con fallimento in tempo finito**

* Dato un programma P, se l’insieme FF(P) (insieme di fallimento finito) denota gli atomi A per cui la dimostrazione fallisce in tempo finito, la regola NF si esprime come: NF(P) = {∼A|A ∈ FF(P)}

**Per risolvere goal generali**, cioè che possono contenere letterali negativi, si introduce un’estensione della risoluzione SLD, nota come **risoluzione SLDNF** [Clark 1978].

* Combina la risoluzione SLD con la negazione per fallimento (NF, Negation as Failure)

SLDNF

Sia :- L1, …, LM il gol (generale) corrente, in cui L1, ..., LM sono letterali (atomi o

negazione di atomi).

* Un passo di risoluzione SLDNF si schematizza come segue:
  + Non si seleziona alcun letterale negativo Li se non è “ground”
  + Se il letterale selezionato Li è positivo, si compie un passo ordinario di risoluzione SLD
  + Se Li è del tipo ∼A (con A “ground”) e A fallisce finitamente (cioè ha un albero SLD di fallimento finito), Li ha successo e si ottiene il nuovo risolvente :- L1, …, Li-1,Li+1, …, Lm.

La risoluzione SLDNF è alla base della realizzazione della negazione per fallimento nei sistemi Prolog

* **Per dimostrare ∼A**, dove A è un atomo, l’interprete del linguaggio **cerca di costruire una dimostrazione per A**
  + **Se la dimostrazione ha successo, allora la dimostrazione di ∼A fallisce**,
  + **Se la dimostrazione per A fallisce finitamente ∼A si considera dimostrato con successo**

**Risolvere con successo un letterale negativo non introduce alcun legame** (unificazione) **per le variabili** dal momento che si considerano solo letterali negativi ”ground”. Al nuovo risolvente: :- L1, …, Li−1, Li+1, …, Lm non si applica alcuna sostituzione

Una **regola di calcolo si dice safe se seleziona un letterale negativo solo quando è ”ground”.** La selezione di letterali negativi solo ”ground” è necessaria per garantire correttezza e completezza della risoluzione SLDNF.

Il linguaggio Prolog seleziona sempre il letterale più a sinistra, senza controllare che sia “ground” e quindi non adotta una regola di selezione safe. Realizzazione non corretta della risoluzione SLDNF.

Esempio:

* Se abbiamo il gol: -capitale(X) si ottiene una risposta sbagliata.
* Scritta in forma logica: ∃X capitale(X)
* Dopodiché si nega il risultato ottenendo ∼(∃X capitale(X))
* Che corrisponde a ∀X(∼capitale(X))
* Quindi se esiste una X che è capitale, fallisce,
  + Ma la query originaria era: ∃X ∼capitale(X)
* Affinché la risposta sia corretta dobbiamo togliere la quantificazione, quindi non avere una variabile ma una costante.

**Il not è indicato dal simbolo \+**

Operazioni sugli insiemi

In Prolog **un insieme è una lista di oggetti senza ripetizioni**.

Alcune operazioni

| Unione di S1 e S2 | union([], S2, S2).  union([X|REST], S2, S) :- member(X, S2), !,  union(REST, S2, S).  union([X|REST], S2, [X|S]) :-  union(REST, S2, S). |
| --- | --- |
| Intersezione di S1 e S2 | intersection([], S2, []).  intersection([H|T], S2, [H|T3]) :-  member(H, S2), !,  intersection(T, S2, T3).  intersection([H|T], S2, S3) :-  intersection(T, S2, S3). |

Meta predicati

Meta-predicati, hanno come argomenti termini che rappresentano parti di programma

**Call(T)**

Il termine T viene trattato come un atomo predicativo e viene richiesta la valutazione del goal T all’interprete Prolog. Serve per richiedere la valutazione per valutare il gol all’interno delle parentesi. Può essere usato per realizzare un costrutto if-then-else:

* if\_then\_else(Cond, Goal1, \_Goal2):- call(Cond), !, call(Goal1).
* if\_then\_else(\_Cond, \_Goal1, Goal2):- call(Goal2).

**Atom(T)**  T è un atomo non numerico

**Number**(T) T è un numero (intero o reale)

**Integer**(T) Tè un numero intero

**Atomic**(T) T è un atomo oppure un numero (ossia T non è un termine composto)

**Var**(T) T è una variabile non istanziata

**Nonvar**(T) T non è una variabile

**Fail**

La valutazione del predicato fail fallisce sempre e quindi forza l’attivazione del meccanismo di backtracking. Vedremo alcuni esempi di uso del predicato fail:

* Per ottenere forme di iterazione sui dati
* Per implementare la negazione per fallimento
* Per realizzare una implicazione logica

Query di secondo ordine

In alcuni casi può essere interessante poter rispondere a query del secondo ordine, ossia a query del tipo “*Quale è l’insieme S di elementi X che soddisfano la query p(X)?*”

**Setof(X,P,S)** Insieme S delle istanze X che soddisfano il gol P

**Bagof(X,P,L)** Lista L delle istanze X che soddisfano il gol P

In entrambi i casi, se non esistono X che soddisfano il goal P i predicati falliscono

bagof produce una lista in cui possono essere contenute ripetizioni, setof produce una lista corrispondente ad un insieme in cui non ci sono ripetizioni.

Esempi:

Dato: padre(mario,paola). padre(mario,aldo). padre(giuseppe,maria).

* setof(X, padre(X,Y), S).
  + Non fornisce tutti gli X per cui padre(X,Y) è vera, ma tutti gli X per cui, per lo stesso valore di Y, padre(X,Y) è vera
* setof(X, Y^padre(X,Y), S).
  + Per quantificare esistenzialmente Y si può usare l’operatore esistenziale ^ (restituisce gli X tali per cui esiste un Y)

**Findall(X,P,S)**

Per ottenere la stessa semantica di bagof con quantificazione esistenziale per la variabile non usata nel primo argomento esiste un predicato predefinito findall(X,P,S) vero se S è la lista delle istanze X per cui la proprietà P è vera

* Se non esiste alcun X per cui P è vera findall non fallisce, ma restituisce una lista vuota (errore se non esiste il predicato chiamato P)