GOLOG

GOLOG è un linguaggio di programmazione ad "altissimo" livello che permette di modellare comportamenti complessi in un mondo che evolve dinamicamente.

Nei linguaggi di programmazione standard i programmi sono sequenze di istruzioni ad alto livello che poi vengono codificate a più basso livello per essere eseguite dalla macchina.

Un programma GOLOG è un'azione complessa, che viene **ridotta ad azioni primitive**, corrispondenti ad azioni "reali" nel dominio di applicazione.

Azioni complesse: sequenza di azioni, test, azioni non deterministiche, azioni condizionali, cicli, procedure.

La semantica delle azioni complesse è definita nel Situation Calculus.

Applicazioni: robotica cognitiva, controllo di robot, programmazione di agenti intelligenti, simulazione di sistemi dinamici.

Sito del gruppo di robotica cognitiva dell'Università di Toronto:

`http://www.cs.toronto.edu/cogrobo/`

Pubblicazioni e codice Prolog di un interprete Golog.

Rappresentazione delle azioni primitive

Le "primitive" del linguaggio sono definite dall'utente: sono le azioni semplici del calcolo delle situazioni.

Le azioni primitive sono definite per mezzo degli assiomi delle precondizioni, assiomi dello stato successore, ecc.

In Prolog (utilizziamo il fluente ulteriore `clear`):

```
%  Direttive per SWI-prolog
:- discontiguous clear/2, on/3, onTable/2.

%  Action Precondition Axioms.
poss(putOn(X,Y),S) :-  clear(X,S), clear(Y,S), \+ on(X,Y,S), \+ X=Y.
poss(putOnTable(X),S) :- clear(X,S), \+ onTable(X,S).

%  Successor State Axioms.
on(X,Y,do(A,S)) :- A = putOn(X,Y) ;
                   on(X,Y,S), \+ (A = putOnTable(X); A = putOn(X,_)).
onTable(X,do(A,S)) :- A = putOnTable(X) ;
                      onTable(X,S), \+ A= putOn(X,_).
clear(X,do(A,S)) :- on(Y,X,S), (A = putOn(Y,_) ; A = putOnTable(Y)) ;
                    clear(X,S), \+ A = putOn(_,X).
```

Osservazioni

Assiomi delle precondizioni:

$$\forall s \forall x \forall y (\neg on(x, y, s) \wedge x \neq y \wedge clear(x, s) \wedge clear(y, s) \rightarrow Poss(PutOn(x, y), s))$$

```
poss(putOn(X,Y),S) :-  clear(X,S), clear(Y,S), \+ on(X,Y,S), \+ X=Y.
```

`Head :- Tail` rappresenta la chiusura universale di $Tail \rightarrow Head$.

Assiomi dello stato successore:

$$\forall a \forall s \forall x \{Poss(a, s) \rightarrow [onTable(x, do(a, s)) \equiv a = putOnTable(x) \vee (onTable(x, s) \wedge \neg \exists y\, a = putOn(x, y))]$$

```
onTable(X,do(A,S)) :- A = putOnTable(X) ;
                      onTable(X,S), \+ A= putOn(X,_).
```

- Si ignora la condizione $Poss(a, s)$
- La doppia implicazione $onTable(x, do(a, s)) \equiv ....$ è sostituita dall'implicazione da destra a sinistra: $.....\rightarrow onTable(x, do(a, s))$.

  Per la semantica del Prolog, però, le due cose si equivalgono.

Rappresentazione della situazione iniziale

```
/* Initial Situation */
onTable(a,s0).
onTable(b,s0).
clear(a,s0).
clear(b,s0).
```

Si specificano soltanto gli atomi positivi (assunzione del mondo chiuso: ciò che non è dimostrabile è falso).

Altri assiomi

In Prolog non si rappresentano:

- gli assiomi del nome unico: nomi e strutture sintatticamente diverse sono necessariamente diversi (non unificabili) in Prolog;
- l'assioma del dominio finito;
- l'assioma di induzione.

Se è necessario tipizzare gli oggetti:

```
robot(pippo).
robot(pluto).
stanza(X) :- member(X,[cucina,salotto,cantina]).
```

(secondo la semantica del Prolog, non vi sono altri robot oltre a pippo e pluto, e le uniche stanze sono la cucina, il salotto e la cantina).

Pianificazione deduttiva in Golog?

```
/* Goal */
goal(S) :- onTable(b,S), on(a,b,S).

?- goal(S).
S = do(putOnTable(b), do(putOn(a, b), do(putOn(b, a), _G241)))
```

- non abbiamo rappresentato l'assioma di induzione: le situazioni non sono solo i termini che si possono costruire a partire dalla situazione iniziale S_0 utilizzando la funzione *do*;
- abbiamo ignorato la condizione di eseguibilità delle azioni.

Si ottiene come soluzione una sequenza di azioni che di fatto non è eseguibile.

Quello che si deve dimostrare, a partire dalla descrizione del dominio e del problema, è che esiste una situazione **raggiungibile dalla situazione iniziale** in cui il goal è vero.

Un piano è una sequenza di azioni

Un piano è un'**azione complessa**: è costituito da una sequenza di **azioni primitive**

- Se a è un'azione primitiva, allora a è un piano
- Se π_1 e π_2 sono piani, allora $[\pi_1 : \pi_2]$ è un piano

```
:- op(950, xfy, [:]).   /* Action sequence */
```

Il predicato Do

Si definisce il simbolo di predicato $Do(\pi, s, s')$ per rappresentare il fatto che

eseguendo il piano π a partire dalla situazione s si raggiunge la situazione s'.

Definizione induttiva di Do

Azioni Primitive: se a è un'azione primitiva (un termine di tipo azione del Situation Calculus), allora:

$$Do(a, s, s') =_{def} Poss(a, s) \wedge s' = do(a, s)$$

Attenzione: *do* è un simbolo di funzione, Do è un simbolo di predicato.

Sequenze: se π_1 e π_2 sono piani, allora:

$$Do([\pi_1; \pi_2], s, s') =_{def} \exists s^* \left(Do(\pi_1, s, s^*) \wedge Do(\pi_2, s^*, s')\right)$$

Una formula della forma $Do(...)$ è un'abbreviazione

Ad esempio, nel mondo dei blocchi:

$$\begin{aligned} & Do([putOnTable(A); putOn(B, A)], s, s') \\ & \quad =_{def} \exists s^* \left(Do(putOnTable(A), s, s^*) \wedge Do(putOn(B, A), s^*, s')\right) \\ & \quad =_{def} \exists s^* \left(Poss(putOnTable(A), s) \wedge s^* = do(putOnTable(A), s)\right. \\ & \qquad \left. \wedge Poss(putOn(B, A), s^*) \wedge s' = do(putOn(B, A), s^*)\right) \end{aligned}$$

Definizione di *Do* in Prolog

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).
do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
```

Si tiene conto delle condizioni di eseguibilità delle azioni.

Attenzione: in Prolog si usa lo stesso simbolo, `do`, per indicare sia il simbolo funzionale che il predicato.

La prima clausola della definizione include la condizione "se a è un'azione primitiva": occorre definire, per ciascun dominio, le azioni primitive.

```
/* definizione delle azioni primitive per il dominio dei blocchi */
primitive_action(putOn(_,_)).
primitive_action(putOnTable(_)).
```

Pianificazione deduttiva

$$Assiomi \vdash \exists \pi \, \exists s \, (Do(\pi, s_0, s) \wedge Goal(s))$$

```
/* ricerca della soluzione */
solution_plan(P) :- do(P,s0,S), goal(S).
```

Soluzione del problema con due blocchi:

```
?- solution_plan(P).
P = putOn(a, b)
```

Un altro problema semplice:

```
/* Initial Situation   a
                       b */
onTable(b,s0).  on(a,b,s0).  clear(a,s0).
/* Goal                b
                       a */
goal(S) :-  on(b,a,S), onTable(a,S), clear(b,S).

?- solution_plan(P).
P = putOnTable(a):putOn(b, a)
```

Ma la strategia di ricerca del Prolog non è completa

```
/* Initial Situation   a
                       b c */

onTable(b,s0).
onTable(c,s0).
on(a,b,s0).
clear(a,s0).
clear(c,s0).

/* Goal                a
                       b
                       c */
goal(S) :- onTable(c,S), on(a,b,S), on(b,c,S), clear(a,S).
```

Il motore di inferenza del Prolog non trova soluzioni.

È necessario controllare la ricerca del piano

Azioni Complesse

Definizione di azioni complesse usando simboli extralogici (while,if,...) che funzionino come *abbreviazioni* (macro da espandere) per espressioni logiche nel linguaggio del Situation Calculus.

Le azioni complesse possono essere **non deterministiche**: diverse esecuzioni della stessa azione possono risultare in situazioni differenti.

Possono essere utilizzate per scrivere programmi nel Situation Calculus.

- **Sequenze**: prima si esegue A e poi B.
- **Azioni Test**: ϕ è vera nella situazione corrente?
- **Azioni non deterministiche**: si esegue A oppure B.
- **Scelta non deterministica dell'argomento dell'azione**: prendere un oggetto (uno qualunque fra quelli che posso prendere).
- **Indeterminismo nel numero di iterazioni**: un'azione viene eseguita più volte.
- **Istruzioni `if-then-else` e cicli `while`**.
- **Procedure**

Per definire il significato delle azioni complesse, si definisce l'abbreviazione $\mathbf{Do}(\delta, \mathbf{s}, \mathbf{s}')$ per rappresentare il fatto che

eseguendo l'azione complessa δ a partire dalla situazione s si può raggiungere la situazione s'.

Definizione induttiva di Do: azioni primitive e sequenze

Azioni Primitive: se a è un'azione primitiva, allora:

$$Do(a, s, s') =_{def} Poss(a, s) \wedge s' = do(a, s)$$

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).
```

Sequenze

$$Do([\delta_1; \delta_2], s, s') =_{def} \exists s^* \, (Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$$

```
do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
```

Definizione induttiva di Do: test

Azioni di Test

$$Do(\phi?, s, s') =_{def} \phi[s] \wedge s = s'$$

Qui ϕ è uno **pseudo-fluente:** si ottiene da una formula del S.C. sopprimendo tutti gli argomenti di tipo situazione (non è una formula del S.C.);
$\phi[s]$ è la formula corrispondente con le situazioni ripristinate (è una formula del S.C.).

Esempio:

$$\phi = \forall x\,(onTable(x) \wedge \neg on(x, A))$$
$$\phi[s] = \forall x\,(onTable(x, s) \wedge \neg on(x, A, s))$$

$$Do(\exists x(onTable(x) \wedge clear(x))?, s, s') =_{def} \exists x(onTable(x, s) \wedge clear(x, s)) \wedge s = s'$$

In Prolog

Sintassi per le azioni di test: `?(F)`, dove `F` è una pseudo-formula:

```
:- op(800, xfy, [&]).    /* Conjunction */
:- op(850, xfy, [v]).    /* Disjunction */
:- op(870, xfy, [=>]).   /* Implication */
:- op(880,xfy, [<=>]).   /* Equivalence */
```

Negazione: segno "meno"
Formule quantificate: `all(Variabile,Formula)` e `some(Variabile,Formula)`.

```
do(?(P),S,S) :- holds(P,S).
```

```
holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
```

```
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- \+ holds(some(V,P),S).  /* Negation */
holds(-P,S) :- isAtom(P), \+ holds(P,S).     /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* The following clause treats the holds predicate for non fluents, including
   Prolog system predicates. For this to work properly, the GOLOG programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
        restoreSitArg(onTable(X),S,onTable(X,S)).              */

holds(A,S) :- restoreSitArg(A,S,F), F ;
              \+ restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- \+ (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).
```

Caso base

```
holds(A,S) :- restoreSitArg(A,S,F), F ;
              \+ restoreSitArg(A,S,F), isAtom(A), A.

%  Restore suppressed situation arguments
restoreSitArg(onTable(X),S,onTable(X,S)).
restoreSitArg(on(X,Y),S,on(X,Y,S)).
restoreSitArg(clear(X),S,clear(X,S)).
```

Esempio: nel mondo dei blocchi, con situazione iniziale:

```
onTable(a,s0).   on(b,a,s0).   clear(b,s0).
```

Per dimostrare il goal:

```
?- holds(onTable(b),do(putOnTable(b),s0))
```

si applica la clausola base della definizione di `holds`, ed il Prolog tenta di soddisfare:

```
restoreSitArg(onTable(b),do(putOnTable(b),s0),F), F
```

Il primo atomo è dimostrabile, con legame `F = onTable(b,do(putOnTable(b),s0))`.
Quindi il Prolog tenta di dimostrare il goal

```
onTable(b, do(putOnTable(b),s0))
```

che ha successo.

Caso base (II)

```
holds(A,S) :- restoreSitArg(A,S,F), F ;
              \+ restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- \+ (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).
```

Se `A` non è uno pseudo-fluente (`restoreSitArg(A,S,F)` fallisce) né una pseudo-formula complessa (`isAtom(A)`), viene invocato `A` come goal Prolog.
Il Golog estende il Prolog: può utilizzare ogni predicato Prolog (indipendente dalla situazione `S`).

```
?- holds(some(x,x is 3+1),s0).
true
```

Quantificatore esistenziale

```
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
```

dove `sub((Name,New,Term1,Term2)`: `Term2` si ottiene da `Term1` sostituendo `Name` con `New`.
Esempio:

```
?-  holds(some(x,onTable(x)),s0)
```

si riduce a:

```
?-  sub(x,_,onTable(x),P1), holds(P1,s0).
```

il primo goal ha successo con (ad esempio): `P1 = onTable(_G176)`.
Quindi:

```
?- holds(onTable(_G176),s0)
```

che si riduce a `onTable(_G176,s0)`, che ha successo.

Negazione

```
holds(-P,S) :- isAtom(P), \+ holds(P,S).     /* Negation by failure */
```

La semantica della negazione è quella della negazione come fallimento.
Perché questa funzioni come la negazione logica, il predicato `holds` **non deve mai essere applicato alla negazione di una pseudo-formula contenente variabili Prolog non legate al momento dell'esecuzione.**

Ad esempio, nella situazione iniziale

```
%  Initial situation
onTable(a,s0).   on(b,a,s0).   clear(b,s0).
```

esiste un blocco che non è sul tavolo, ma:

```
?- holds(some(x,-onTable(x)),s0).
false.
```

```
holds(some(x,-onTable(x)),s0)  ⇒  holds(-onTable(X),s0)
                               ⇒  \+ holds(onTable(X),s0)
```

`holds` è invocato sulla negazione di una pseudo-formula contenente una variabile Prolog (`X`) non legata.

`holds(onTable(X),s0)` ha successo (il blocco `a` è sul tavolo), quindi `\+ holds(onTable(X),s0)` fallisce e così anche `holds(some(x,-onTable(x)),s0)`.

Quantificatore universale

Attenzione anche quando si utilizza il quantificatore universale o la negazione di un esistenziale:

```
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- \+ holds(some(V,P),S).
holds(all(V,P),S) :- holds(-some(V,-P),S).

?- holds(-all(x,onTable(x)),s0).
false.
?- holds(all(x,onTable(x)),s0).
true
```

Un possibile modo di risolvere il problema: utilizzare soltanto formule con quantificazione limitata

$$\forall x\,(T(x) \rightarrow A(x)) \quad \text{e} \quad \exists x\,(T(x) \wedge A(x))$$

dove $T(x)$ è la formula atomica che rappresenta il tipo di x.

```
block(a).
block(b).

?- holds(some(x,block(x) & -onTable(x)),s0).
true
?- holds(all(x,block(x) => onTable(x)),s0).
false.
```

Definizione induttiva di Do: scelte non deterministiche

Scelta non deterministica fra due azioni

$$Do((\delta_1|\delta_2), s, s') =_{def} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

```
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
```

$$\begin{aligned}
& Do((putOnTable(a) \mid putOnTable(b)), s, s') \\
\equiv_{def}\ & Do(putOnTable(a), s, s') \vee Do(putOnTable(b), s, s') \\
\equiv_{def}\ & (Poss(putOnTable(a), s) \wedge s' = do(putOnTable(a), s)) \vee \\
& (Poss(putOnTable(b), s) \wedge s' = do(putOnTable(b), s))
\end{aligned}$$

Se si ha:

```
%  Initial situation
onTable(a,s0).   on(b,a,s0).   clear(b,s0).
```

Ci sono due modi di soddisfare il goal `do(?(onTable(a)) # putOnTable(b),s0,S)`:

```
?- do(?(onTable(a)) # putOnTable(b),s0,S).
S = s0 ;
S = do(putOnTable(b), s0) ;
false.
```

Scelta non deterministica degli argomenti dell'azione

$$Do((\pi x)\delta(x)), s, s') =_{def} \exists x\, Do(\delta(x), s, s')$$

Esempio: Un robot si può muovere do x a y con l'azione *goto(x,y)*. L'azione complessa $\gamma = (\pi x)\ goto(r, x)$ rappresenta l'azione del robot di muoversi in un luogo qualunque $x,\ x \neq r$.

$$\begin{aligned} Do((\pi x)\, goto(r, x), s, s') &=_{def} \exists x\, Do(goto(r, x), s, s') \\ &=_{def} \exists x\, (Poss(goto(r, x), s) \wedge s' = do(goto(r, x), s)) \end{aligned}$$

$$\begin{aligned} Do((\pi x)\, putOnTable(x), s, s') &=_{def} \exists x\, Do(putOnTable(x), s, s') \\ &=_{def} \exists x\, (Poss(putOnTable(x), s) \wedge s' = do(putOnTable(x), s)) \end{aligned}$$

In Prolog:

```
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */

%  Initial situation
onTable(a,s0).   on(b,a,s0).   clear(b,s0).
onTable(c,s0).   on(d,c,s0).   clear(d,s0).

?- do(pi(x,putOnTable(x)),s0,S).
S = do(putOnTable(b), s0) ;
S = do(putOnTable(d), s0) ;
false.
```

Definizione induttiva di Do: iterazione

Iterazione non deterministica:

$Do(\delta^*, s, s')$: esecuzione di δ zero o più volte.

$Do(\delta^*, s, s')$ è la chiusura riflessiva e transitiva di $Do(\delta, s, s')$.

Ma **la chiusura transitiva R* di un relazione R non è definibile in logica del primo ordine**.

R^* si può definire induttivamente:

- se $R(x, y)$ allora $R^*(x, y)$;
- se, per qualche z,$R(x, z)$ e $R^*(z, y)$, allora $R^*(x, y)$; di y;
- nessun'altra coppia (x, y) è nella relazione R^* (clausola di chiusura).

In generale, non sono definibili in logica del primo ordine i predicati o relazioni definite induttivamente.

Attenzione

$$(1) \quad \forall x \forall y(antenato(x, y) \equiv genitore(x, y) \vee \exists z(genitore(x, z) \wedge antenato(z, y)))$$

non "definisce" *antenato*: consideriamo, ad esempio, $\mathcal{M}$ con $D = \{a, b\}$, $\mathcal{M}(genitore) = \{(a, a), (b, b)\}$ e $\mathcal{M}(antenato) = \{(a, a), (b, b), (a, b)\}$; $\mathcal{M} \models 1$, ma $\mathcal{M}(antenato)$ non è la chiusura transitiva di $\mathcal{M}(genitore)$.

Quello che vorremmo dire è che *antenato* è **la più piccola relazione** che soddisfa (1)

Definizione induttiva di Do: iterazione (II)

Per definire la chiusura (riflessiva e) transitiva di una relazione occorre la **logica del secondo ordine**, in cui si può quantificare sui predicati

$$Do(\delta^*, s, s') =_{def} \forall P \{ \forall s_1 P(s_1, s_1) \wedge \forall s_1, s_2, s_3 [P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \rightarrow P(s_1, s_3)] \rightarrow P(s, s') \}$$

L'esecuzione di δ zero o più volte condurrà dalla situazione s alla situazione s' sse (s, s') appartiene a ogni insieme P (dunque al più piccolo insieme P) tale che:

1. per ogni situazione s_1, $(s_1, s_1) \in P$.
2. per ogni coppia di situazioni (s_1, s_2), se $(s_1, s_2) \in P$ e l'esecuzione di δ in s_2 porta in s_3, allora $(s_1, s_3) \in P$.

Questa è la definizione standard in **logica del secondo ordine** della chiusura riflessiva e transitiva di un insieme.

Definizione della chiusura transitiva di una relazione in Prolog

```
antentato(X,Y) :- genitore(X,Y).
antentato(X,Y) :- genitore(X,Z), antenato(Z,Y).
```

La semantica di questa definizione è proprio la formula

(1) $\forall x \forall y(antenato(x,y) \equiv genitore(x,y) \vee \exists z(genitore(x,z) \wedge antenato(z,y)))$

Tutte le istanze positive di $antenato(x,y)$ sono conseguenze logiche di (1).

Ma non necessariamente lo sono quelle negative (della forma $\neg\, antenato(x,y)$).

Si aggiunga la seguente definizione della relazione `genitore`:

```
genitore(a,a)
genitore(b,b)
```

E si provi ad eseguire il goal

```
?- \+ antenato(a,b)
```

Definizione Prolog dell'iterazione

```
do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).
```

Istruzioni condizionali e cicli

Istruzioni condizionali e cicli while possono essere definiti usando i costrutti precedenti.

Istruzione condizionale `if then else`

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \quad =_{def} \quad [\phi?; \delta_1] \mid [\neg\phi?; \delta_2]$$

Esecuzione non deterministica della sequenza $[\phi?; \delta_1]$, in cui si ha l'esecuzione dell'azione test $\phi?$ seguita dall'esecuzione di δ_1, oppure della sequenza $[\neg\phi?; \delta_1]$, in cui si ha l'esecuzione dell'azione test $\neg\phi?$ seguita dall'esecuzione di δ_2.

```
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(-P) : E2),S,S1).
```

Istruzione di ciclo `while do`

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \quad =_{def} \quad [\ [\phi?; \delta]^*\ ; \neg\phi?]$$

Sequenza dell'istruzione con numero non determinato di iterazioni della sequenza $[\phi?; \delta]$ (azione test ϕ e azione δ) e dell'azione test $\neg\phi?$.

```
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(-P),S,S1).
```

Esempio

```
%  Initial situation
onTable(a,s0).   on(b,a,s0).   on(c,b,s0).   clear(c,s0).
onTable(d,s0).   on(e,d,s0).   clear(e,s0).
```

Aggiungiamo la definizione del predicato `block`:

```
block(X) :- member(X,[a,b,c,d,e]).
```

Finché è possibile, sposta un blocco sul tavolo:

```
?- do(while(some(x, block(x) & -onTable(x)),
            pi(x,putOnTable(x))),s0,S).
S = do(putOnTable(e), do(putOnTable(b), do(putOnTable(c), s0)))
```

Procedure

Dichiarazione di una procedura, con parametri formali $v_1, ..., v_n$ e corpo δ:

$$\textbf{proc } P\ (v_1, ..., v_n)\ \delta \textbf{ endProc}$$

Chiamata di procedura in un programma:

$$P(t_1, ..., t_n)$$

Per ogni simbolo di predicato P con arità $n + 2$ si definisce:

$$Do(P(t_1, \ldots, t_n), s, s') =_{def} P(t_1, \ldots, t_n, s, s')$$

Eseguendo la procedura P sui parametri attuali $t_1, \ldots, t_n$ si ottiene la transizione dalla situazione s alla situazione s'.

Anche la semantica di procedure **ricorsive** nel Situation Calculus richiede la **logica del secondo ordine**.

se $P(v_1, ..., v_n)$ è una procedura definita con corpo $E(v_1, ..., v_n)$:

$$\textbf{proc } P(v_1, ..., v_n)\ E(v_1, ..., v_n) \textbf{ endProc}$$

allora $Do(P(t_1, ..., t_n), s, s')$ è la più piccola relazione binaria chiusa rispetto alla relazione $Do(E(t_1, ..., t_n), s, s')$.

Procedure in Prolog

Una procedura è definita mediante un fatto della forma `proc(Testa,Corpo)`.

```
proc(p(V1,...,Vn), corpo(V1,...,Vn)).
```

Definizione di `do` per la chiamata di procedure:

```
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
```

Esempio

```
block(X) :- member(X,[a,b,c,d,e]).

%  Initial situation
onTable(a,s0).   on(b,a,s0).   on(c,b,s0).   clear(c,s0).
onTable(d,s0).   on(e,d,s0).   clear(e,s0).

proc(smontaTutti,
     while(some(x, block(x) & -onTable(x)),
   pi(x,putOnTable(x)))).

?- do(smontaTutti,s0,S).
S = do(putOnTable(e), do(putOnTable(b), do(putOnTable(c), s0))) ;
S = do(putOnTable(b), do(putOnTable(e), do(putOnTable(c), s0))) ;
.....

proc(smonta(X),
     ?(onTable(X))#
    pi(y,?(on(X,y)):
      putOnTable(X): smonta(y))).

?- do(smonta(c),s0,S).
S = do(putOnTable(b), do(putOnTable(c), s0))
```

Esempio: controllo di un ascensore

```
        %  THE SIMPLIFIED ELEVATOR CONTROLLER

:- [golog_swi].
:- discontiguous currentFloor/2, on/2.

% Primitive control actions
primitive_action(up).   % Move elevator 1 floor up
primitive_action(down). % Move elevator 1 floor down
primitive_action(open).        % Open elevator door.
primitive_action(close).       % Close elevator door.

% lower and higher floors
top(10).
bottom(-2).

% Preconditions for Primitive Actions.
poss(up,S) :-
        currentFloor(M,S), top(Top), M<Top.
poss(down,S) :-
        currentFloor(M,S), bottom(Bot), M > Bot.
poss(open,_).
poss(close,_).
```

Esempio (II)

```
% Successor State Axioms for Primitive Fluents.

% currentFloor(M,S) = the elevator is at floor S in situation S
currentFloor(M,do(A,S)) :-
        (A=up, currentFloor(N,S), M is N+1) ;
        (A=down, currentFloor(N,S), M is N-1) ;
        (currentFloor(M,S), not(A=up), not(A=down)).

% on(M,S): the call button M is on in situation S
on(M,do(_,S)) :- on(M,S), not(currentFloor(M,S)).

% Initial Situation. Call buttons: 3 and 5. The elevator is at floor 4.
on(3,s0).
on(5,s0).
currentFloor(4,s0).

% Restore suppressed situation arguments.
restoreSitArg(on(N),S,on(N,S)).
restoreSitArg(currentFloor(M),S,currentFloor(M,S)).
```

Esempio (III): procedure

```
% Procedure down(n): move the
% elevator down n floors
proc(down(N),
     ?(N=0) #
     ?(N>0) : down :
      pi(m, ?(m is N-1) : down(m))).
```

```
% Procedure up(n): move the
% elevator up n floors
proc(up(N),
     ?(N=0) #
     ?(N>0) : up :
      pi(m, ?(m is N-1) : up(m))).
```

```
/* ----------------------------
?- do(up(4),s0,S),
       currentFloor(F,S).
S = do(up, do(up,
             do(up, do(up, s0))))
F = 8

?- do(up(10),s0,S).
No
```

```
/* ---------------------------
?- do(down(4),s0,S),
         currentFloor(F,S).
S = do(down, do(down,
                do(down, do(down, s0))))
F = 0

?- do(down(10),s0,S).
No
```

Esempio (IV)

```
% Procedure goFloor(n): go to floor n
proc(goFloor(N),
     ?(currentFloor(N)) #
     pi(n1, ?(currentFloor(n1)) :
            if(n1<N, up, down))
     : goFloor(N)).

/* ---------------------------------------
?- do(goFloor(6),s0,S).
S = do(up, do(up, s0))

?- do(goFloor(0),s0,S).
S = do(down, do(down, do(down, do(down, s0))))
---------------------------------------- */
```

Esempio (V)

```
proc(serve(N), goFloor(N) : open : close).
proc(serveAfloor, pi(n, ?(on(n)) : serve(n))).

/* ------------------------------------

?- do(serve(6),s0,S).
S = do(close, do(open, do(up, do(up, s0))))

?- do(serveAfloor,s0,S).
S = do(close, do(open, do(down, s0)))

------------------------------------- */

proc(main,
     while(some(x,on(x)),serveAfloor)).

/* ------------------------------------

?- do(main,s0,S).
S = do(close, do(open, do(up, do(up, do(close, do(open, do(down,s0)))))))
```

Esempio 2

```
/* Definizione azioni primitive */
primitive_action(pickup(_)).
primitive_action(putOnTable(_)).
primitive_action(putOnFloor(_)).

/* Assiomi delle precondizioni per le azioni primitive */
poss(pickup(_),S) :- armempty(S).
poss(putOnTable(X),S) :- holding(X,S).
poss(putOnFloor(X),S) :- holding(X,S).

/* Assiomi di stato successore per i fluenti primitivi*/
holding(X,do(A,S)) :- A=pickup(X);
                      holding(X,S), \+ A=putOnTable(X), \+ A=putOnFloor(X).
onTable(X,do(A,S)) :- A=putOnTable(X); onTable(X,S), \+ A=pickup(X).
onFloor(X,do(A,S)) :- A=putOnFloor(X); onFloor(X,S), \+ A=pickup(X).
armempty(do(A,S))  :- A=putOnTable(X); A=putOnFloor(X);
                      armempty(S), \+ A=pickup(X).
```

Esempio 2 (II)

```
/* Procedure */
proc(clearTable, while(some(y,onTable(y)),
                       pi(x,(?(onTable(x)) : removeFromTable(x))))).
proc(removeFromTable(X), pickup(X) : putOnFloor(X)).

/* Stato iniziale */
onTable(b,s0). onTable(a,s0). armempty(s0).

/* Regole per ripristinare gli argomenti situazione soppressi*/
restoreSitArg(onTable(X),S,onTable(X,S)).
restoreSitArg(onFloor(X),S,onFloor(X,S)).
restoreSitArg(holding(X),S,holding(X,S)).
restoreSitArg(armempty,S,armempty(S)).

/* ------------------------------------------

?- do(clearTable,s0,S).
S = do(putOnFloor(a), do(pickup(a), do(putOnFloor(b), do(pickup(b), s0))))
```

Golog: conclusioni

Linguaggio di programmazione logica per implementare applicazioni in domini dinamici (robotica, controllo di processi, agenti software intelligenti, ecc.). È basato su una **teoria formale delle azioni** (Situation Calculus esteso).

Caratteristiche principali

- Un programma Golog è una **macro** che si espande (durante la valutazione) in una **formula del Situation Calculus** che coinvolge solo azioni primitive e fluenti.
- I programmi Golog sono **valutati per ottenere un legame per la variabile situazione quantificata esistenzialmente nella chiamata principale**:

  $$\exists s\, Do(program, S_0, s)$$

  Il legame ottenuto è una traccia simbolica dell'esecuzione del programma e denota la sequenza di azioni da eseguire.
- Le azioni primitive e i fluenti sono definiti dal programmatore, mediante gli assiomi delle precondizioni e gli assiomi dello stato successore.
- Il programmatore Golog definisce azioni complesse come **"schemi"**, senza specificare nel dettaglio come devono essere eseguige. È compito del theorem prover generare le sequenze di azioni primitive direttamente eseguibili dall'esecutore.

Esercizi

1. Scrivere le formule del Calcolo delle Situazioni che costituiscano la definizione di:
   (a) $Do([go(a,b);(take(obj1) \mid take(obj2))], s, s')$
   (b) $Do((\pi x)[\neg onTable(x)?; putOnTable(x)], s, s')$
   (c) $Do(\textbf{if } onTable(a) \textbf{ then } putOn(a,b) \textbf{ else } putOnTable(a) \textbf{ endIf}, s, s')$

   (dove s e s' sono variabili, $onTable$ è un fluente, go, $take$, $putOnTable$ e $putOn$ sono azioni primitive);
2. Nel mondo dei blocchi, definire:
   (a) una procedura $tower(x,n)$, che mette n blocchi sopra (la torre che ha in cima) il blocco x;
   (b) una procedura $make\_tower(n)$, che costruisce una torre di n blocchi.
3. Un robot si può muovere tra diversi posti e comprare oggetti. Per comprare un oggetto deve essere in un posto in cui si vende tale oggetto. Dopo aver comprato un oggetto, il robot lo possiede. Descrivere tale dominio in Prolog e scrivere una procedura Golog $go\_and\_buy(x)$ che conduca il robot ad andare in un posto in cui si vende x e comprarlo.
4. Un robot agisce in un ambiente costituito da un certo numero di locazioni, e dispone di una borsa che può trasportare oggetti. Si assume che la borsa sia di capienza illimitata. Le azioni che può compiere sono andare da una locazione all'altra, mettere un oggetto nella borsa, togliere un oggetto dalla borsa. Descrivere tale dominio in Prolog e scrivere una procedura Golog $bring(x, from, to)$ con la quale il robot va a prendere l'oggetto x che si trova in $from$, e lo porta da $from$ a to.