Il Calcolo delle Situazioni e il linguaggio Golog

a.a. 2010/2011

Marta Cialdea Mayer

Indice

Capitolo 1

Preliminari

1.1 Teorie con uguaglianza

I linguaggi della logica dei predicati con uguaglianza includono tra i simboli non logici il predicato a due argomenti $=$ (utilizzato in forma infissa), la cui semantica è l'identità: $\{(d, d) \mid d \in D\}$, dove D è il dominio dell'interpretazione.

Gli assiomi propri di una teoria con uguaglianza includono quelli specifici per essa:

1. $\forall x(x = x)$ (riflessività)
2. $\forall x \forall y(x = y \rightarrow (A \rightarrow A'))$
   dove A' si ottiene da A sostituendo alcune (non necessariamente tutte) occorrenze libere di x con y.

Da tali assiomi sono derivabili, ad esempio:

$$\forall x \forall y(x = y \rightarrow y = x)$$
$$\forall x \forall y \forall z(x = y \wedge y = z \rightarrow x = z)$$

In queste dispense, utilizzeremo la notazione $t \neq t'$ come abbreviazione della formula $\neg(t = t')$.

1.2 Logiche a molte sorte

I linguaggi logici a molte sorte (*multi-sorted*) sono analoghi ai linguaggi di programmazione con tipi, e in queste dispense utilizzeremo il termine *tipo* come sinonimo di *sorta*. In tali linguaggi vi sono termini e variabili di *sorte* (tipi) diversi, e il dominio dell'interpretazione è partizionato in sotto-domini disgiunti, uno per ogni sorta. Nella semantica di un linguaggio logico con tipi l'interpretazione di ciascuna variabile o termine di tipo σ è un elemento del dominio appartenente al sotto-dominio corrispondente a σ.

Ogni simbolo di predicato (eccetto l'uguaglianza, se presente) o di funzione, inoltre, può essere applicato soltanto ad argomenti di tipi specifici, e ogni termine della forma $f(t_1, ..., t_n)$ ha il proprio tipo.

In maggior dettaglio, i simboli di un linguaggio con tipi includono, oltre agli operatori logici e ai simboli separatori:

- per ogni tipo σ, un insieme infinito di variabili di tipo σ: $x_0^\sigma, x_1^\sigma, ...$;
- per ogni tipo σ, un insieme (eventualmente vuoto) di costanti di tipo σ;
- un insieme (eventualmente vuoto) di simboli di funzione, $f_0, f_1, ...$, a ciascuno dei quali è associata l'informazione sul numero e tipo degli argomenti e sul tipo del valore:
$$f_i : \sigma_1 \times .... \times \sigma_n \rightarrow \sigma$$
- un insieme di simboli di predicato, $p_0, p_1, ...$, a ciascuno dei quali è associata l'informazione sul numero e tipo degli argomenti:
$$p_i : \sigma_1 \times .... \times \sigma_n$$

Ad ogni termine t è associata un unico tipo σ (e scriveremo allora $t : \sigma$), come segue:

- ogni variabile di tipo σ è un termine di tipo σ;
- ogni costante di tipo σ è un termine di tipo σ;
- se $f : \sigma_1 \times .... \times \sigma_n \rightarrow \sigma$ è un simboli di funzione e $t_1 : \sigma_1, ..., t_n : \sigma_n$ sono termini dei tipi specificati, allora $f(t_1, ..., t_n)$ è un termine di tipo σ.

Le formule atomiche sono definite come segue:

- se t e t' sono termini dello stesso tipo, allora $t = t'$ è una formula atomica (se il linguaggio include l'uguaglianza);
- se $p : \sigma_1 \times .... \times \sigma_n$ è un simbolo di predicato e $t_1 : \sigma_1, ..., t_n : \sigma_n$ sono termini dei tipi specificati, allora $p(t_1, ..., t_n)$ è una formula atomica.

Il dominio di un'interpretazione è costituito da un insieme non vuoto D^σ per ogni tipo σ. I sotto-domini devono essere disgiunti: per ogni tipo σ e σ', si deve avere $D^\sigma \cap D^{\sigma'} = \emptyset$.

La semantica delle costanti e dei simboli di predicato e di funzione devono rispettare i tipi dei simboli:

- se $c : \sigma$ è una costante di tipo σ, la sua interpretazione è un oggetto di D^σ;
- se $f : \sigma_1 \times .... \times \sigma_n \rightarrow \sigma$ è un simbolo di funzione, allora la sua interpretazione è una funzione F di tipo $D^{\sigma_1} \times .... \times D^{\sigma_n} \rightarrow D^\sigma$;
- se $p : \sigma_1 \times .... \times \sigma_n$ è un simbolo di predicato, la sua interpretazione è un sottoinsieme di $D^{\sigma_1} \times .... \times D^{\sigma_n}$.

Un'assegnazione di valori alle variabili è una funzione che assegna a ogni variabile di tipo σ un oggetto di D^σ. Di conseguenza, un quantificatore della forma $\forall x^\sigma$ o $\exists x^\sigma$ "quantifica" sul sotto-dominio D^σ:

- $(\mathcal{M}, s) \models \forall x^\sigma A$ se e solo se per ogni oggetto $d \in D^\sigma$, $(\mathcal{M}, s[d/x^\sigma]) \models A$;
- $(\mathcal{M}, s) \models \exists x^\sigma A$ se e solo se esiste un oggetto $d \in D^\sigma$ tale che $(\mathcal{M}, s[d/x^\sigma]) \models A$.

Le logiche a molte sorte non sono in realtà più espressive della logica abituale, senza tipi. Un linguaggio a molte sorte si può infatti ridurre al linguaggio abituale, introducendo un simbolo di predicato unario Q_σ per ogni tipo σ: il significato inteso di $Q_\sigma(t)$ è che il termine t è di tipo σ.

Si può allora trasformare ogni formula del linguaggio a molte sorte in una formula del linguaggio standard come segue: si sostituisce ogni sottoformula della forma $\forall x^\sigma A(x)$ con $\forall x\,(Q_\sigma(x) \rightarrow A(x))$ e ogni formula della forma $\exists x^\sigma A(x)$ con $\exists x\,(Q_\sigma(x) \wedge A(x))$. Se A è una formula del linguaggio a molte sorte, indichiamo con A^* la formula del linguaggio abituale che si ottiene con questa trasformazione; e analogamente per un insieme S di formule: $S^* = \{A^* \mid A \in S\}$.

Si definisce inoltre il seguente insieme T di formule (nel linguaggio senza tipi):

- $\exists x\, Q_\sigma(x)$, per ogni tipo σ;
- $\forall x_1 ... \forall x_n\,(Q_{\sigma_1}(x_1) \wedge ... \wedge Q_{\sigma_n}(x_n) \rightarrow Q_\sigma(f(x_1, ..., x_n)))$, per ogni simbolo di funzione $f : \sigma_1 \times .... \times \sigma_n \rightarrow \sigma$.

Si può allora dimostrare che per ogni insieme $S \cup \{A\}$ di formule del linguaggio a molte sorte: $S \models A$ se e solo se $S^* \cup T \models A^*$.

1.3 Logica del secondo ordine

1.3.1 Limiti della logica del primo ordine

La logica del primo ordine ha alcuni significativi limiti di espressività. Ad esempio, non è possibile definire in logica del primo ordine la chiusura transitiva di una relazione: se R è una relazione binaria, la sua chiusura transitiva R^* è l'insieme di tutte le coppie di elementi (a, b) per i quali esiste una sequenza $a_0, ..., a_n$ tale che: $a = a_0$, $b = a_n$ e per ogni $i = 1, ..., n$, $(a_{i-1}, a_i) \in R$.

Assumiamo ora che la relazione R sia rappresentata da un simbolo di predicato r. Dare una definizione esplicita di R^* significa identificare una formula $F(x, y)$, con variabili libere x e y, tale che per ogni interpretazione $\mathcal{M}$ che assegna un significato a r (assumiamo $\mathcal{M}(r) = R$), e per ogni assegnazione s: $(\mathcal{M}, s) \models F(x, y)$ se e solo se la coppia $(s(x), s(y))$ appartiene alla chiusura transitiva R^* di R. In altri termini, l'insieme delle coppie che appartengono alla chiusura transitiva della relazione R deve essere esattamente l'insieme delle coppie che soddisfano la formula $F(x, y)$.

Se si può trovare una formula $F(x, y)$ con tali caratteristiche, si può allora "definire" un nuovo simbolo di predicato r^*, per rappresentare la chiusura transitiva di r, mediante la formula:

$$\forall x \forall y\,(r^*(x, y) \equiv F(x, y))$$

Si può tuttavia dimostrare che non esiste una tale formula $F(x, y)$ in *logica del primo ordine*.

Tipicamente, non sono definibili al primo ordine le relazioni che vengono abitualmente definite induttivamente, nella forma "R è il più piccolo insieme tale che ...". La chiusura transitiva di una relazione R è infatti un caso particolare di relazione che si può definire induttivamente. Quella che segue è la sua definizione abituale.

1. se $(a, b) \in R$ allora $(a, b) \in R^*$;

2. se $(a, b) \in R$ e $(b, c) \in R^*$, allora $(a, c) \in R^*$;

3. nient'altro è in R^*.

L'ultima clausola (la *clausola di chiusura*) viene a volte omessa in quanto sottintesa. Un modo alternativo di formulare la definizione induttiva è il seguente: la chiusura transitiva R^* della relazione R è il più piccolo insieme S che soddisfa le seguenti proprietà (o è l'intersezione di tutti gli insiemi S che soddisfano le seguenti proprietà):

1. se $(a, b) \in R$ allora $(a, b) \in S$;

2. se $(a, b) \in R$ e $(b, c) \in S$, allora $(a, c) \in S$.

Osservazione 1 *Occorre prestare attenzione a non considerare una "definizione" della chiusura transitiva di r la formula seguente:*

$$(1) \quad \forall x \forall y \, (r^*(x, y) \equiv r(x, y) \vee \exists z \, (r(x, z) \wedge r^*(z, y)))$$

Tale formula esprime infatti una proprietà *di r^*, ma non è una definizione, in quanto possono esistere diverse relazioni che la soddisfano. Si consideri ad esempio la semplice interpretazione $\mathcal{M}$ con dominio $D = \{a, b\}$, in cui $\mathcal{M}(r) = \{(a, a), (b, b)\}$ e $\mathcal{M}(r^*) = \{((a, a), (b, b), (a, b)\}$. Tale interpretazione è un modello della formula* 1*, ma $\mathcal{M}(r^*)$ non è la chiusura transitiva di $\mathcal{M}(r)$. Il fatto è che la formula* 1 *non cattura il fatto che r^* è* la più piccola *relazione con la proprietà data.*

Quel che vale è soltanto questo: sia $\mathcal{M}$ un qualsiasi modello della formula 1*, con $\mathcal{M}(r) = R$. Allora, per ogni coppia (d, d') di elementi del dominio che appartiene alla chiusura transitiva R^* di R e per ogni assegnazione s, $(\mathcal{M}, s[d/x, d'/y]) \models r^*(x, y)$. Ma non è detto che se $(d, d') \notin R^*$ allora $(\mathcal{M}, s[d/x, d'/y]) \not\models r^*(x, y)$.*

Come già detto, non è possibile, in generale, caratterizzare in logica del primo ordine gli insiemi definiti induttivamente. Consideriamo, come ulteriore esempio, la seguente definizione induttiva dei numeri naturali: $\mathbb{N}$ è il più piccolo insieme tale che:

- $0 \in \mathbb{N}$;

- per ogni n, se $n \in \mathbb{N}$ allora $succ(n) \in \mathbb{N}$.

E si consideri la seguente formula del primo ordine:

$$(2) \quad \forall x \, (nat(x) \equiv x = 0 \vee \exists y \, (nat(y) \wedge x = succ(y)))$$

Analogamente al caso della chiusura transitiva, la formula 2 non "definisce" $\mathbb{N}$: consideriamo ad esempio l'interpretazione $\mathcal{M}$ con dominio $\mathbb{N} \cup \{\infty\}$, in cui $\mathcal{M}(succ)(n) = n+1$ se $n \in \mathbb{N}$ e $\mathcal{M}(succ)(\infty) = \infty$; inoltre $\mathcal{M}(nat) = \mathbb{N} \cup \{\infty\}$. Anche in questo caso, la formula 2 è vera in $\mathcal{M}$, sebbene $\mathcal{M}(nat) \neq \mathbb{N}$.

1.3.2 Sintassi e semantica della logica del secondo ordine

La chiusura transitiva di una relazione e le definizioni induttive, sebbene non definibili nella logica del primo ordine, si possono invece definire nella *logica del secondo ordine*, che consente di quantificare su predicati e funzioni.

La logica del secondo ordine si ottiene aggiungendo ai simboli della logica dei predicati:

- *variabili predicative*: per ogni $n \geq 0$, un insieme infinito di variabili per predicati a n argomenti;
- *variabili funzionali*: per ogni $n \geq 0$, un insieme infinito di variabili per funzioni a n argomenti.

Useremo i simboli X, Y, Z per indicare variabili funzionali o predicative. Le variabili della logica del primo ordine sono ora chiamate *variabili individuali*.

Nella formazione dei termini possono essere utilizzate variabili funzionali (se X è una variabile funzionale a n argomenti e $t_1, ..., t_n$ sono termini, allora $X(t_1, ..., t_n)$ è un termine) e le formule atomiche includono quelle formate utilizzando le variabili predicative (se X è una variabile predicativa a n argomenti e $t_1, ..., t_n$ sono termini, allora $X(t_1, ..., t_n)$ è una formula atomica). Le formule includono quelle che si possono formare utilizzando la quantificazione su variabili funzionali e predicative: se A è una formula e X una variabile funzionale o predicativa, allora $\forall X\, A$ e $\exists X\, A$ sono formule.

Per definire la semantica occorre estendere l'interpretazione delle variabili alle variabili funzionali e predicative: un'assegnazione di valore alle variabili s sul dominio D è una funzione che associa:

- a ogni variabile individuale un oggetto del dominio;
- a ogni variabile funzionale a n argomenti una funzione a n argomenti sul dominio;
- a ogni variabile predicativa a n argomenti un sottoinsieme di D^n.

Inoltre, si definiscono le varianti di un'assegnazione anche per i valori assegnati a variabili predicative o funzionali; sia s un'assegnazione:

- se X è una variabile predicativa a n argomenti, e R una relazione n-aria sul dominio, allora $s[R/X]$ è la variante di s tale che per ogni variabile (individuale, predicativa o funzionale) Y diversa da P, $s[R/P](Y) = s(Y)$, e $s[R/X](X) = R$;
- se X è una variabile funzionale a n argomenti, e F una funzione a n argomenti sul dominio, allora $s[F/X]$ è la variante di s tale che per ogni variabile (individuale, predicativa o funzionale) Y diversa da X, $s[F/X](Y) = s(Y)$, e $s[F/X](X) = F$.

La definizione di interpretazione di un termine secondo l'assegnazione s è estesa stabilendo che: se X è una variabile funzionale, allora $\overline{s}(X(t_1, ..., t_n)) = s(X)(\overline{s}(t_1), ..., \overline{s}(t_n))$.

La definizione della relazione $(\mathcal{M}, s) \models A$ viene estesa con l'aggiunta di una nuova clausola base e delle clausole per la quantificazione su variabili funzionali e predicative:

1. se X è una variabile predicativa, allora $(\mathcal{M}, s) \models X(t_1, ..., t_n)$ sse $\langle \overline{s}(t_1), ..., \overline{s}(t_n) \rangle \in s(X)$.

2. Se X è una variabile predicativa a n argomenti, allora:

   (a) $(\mathcal{M}, s) \models \forall X\, A$ sse per ogni relazione n-aria R sul dominio, $(\mathcal{M}, s[R/X]) \models A$;

   (b) $(\mathcal{M}, s) \models \exists X\, A$ sse esiste una relazione n-aria R sul dominio, tale che $(\mathcal{M}, s[R/X]) \models A$;

3. Se X è una variabile funzionale a n argomenti, allora:

   (a) $(\mathcal{M}, s) \models \forall X\, A$ sse per ogni funzione n-aria F sul dominio, $(\mathcal{M}, s[F/X]) \models A$;

   (b) $(\mathcal{M}, s) \models \exists X\, A$ sse esiste una funzione n-aria F sul dominio, tale che $(\mathcal{M}, s[F/X]) \models A$.

1.3.3 Definizioni induttive in logica del secondo ordine

Consideriamo di nuovo la definizione induttiva dei numeri naturali: $\mathbb{N}$ è il più piccolo insieme tale che:

- $0 \in \mathbb{N}$;
- per ogni n, se $n \in \mathbb{N}$ allora $succ(n) \in \mathbb{N}$.

O, equivalentemente: $\mathbb{N}$ è l'insieme di tutti gli elementi che appartengono a ogni insieme P tale che ($\mathbb{N}$ è l'intersezione di tutti gli insiemi P tali che):

- $0 \in P$;
- per ogni n, se $n \in P$ allora $succ(n) \in P$.

La formula del secondo ordine seguente definisce il predicato nat come l'appartenenza a $\mathbb{N}$:

$$\forall x\, (nat(x) \equiv \forall P[P(0) \wedge \forall y\, (P(y) \rightarrow P(succ(y)))) \rightarrow P(x)]$$

x è un naturale se e solo se appartiene ($P(x)$) a ogni insieme P che contiene 0 e i successori di tutti i suoi elementi ($P(0) \wedge \forall y\, (P(y) \rightarrow P(succ(y)))$).

Si noti che l'assioma di induzione per i numeri naturali, che si può formulare in logica del secondo ordine come segue:

$$\forall P\, [P(0) \wedge \forall x(P(x) \rightarrow P(succ(x)) \rightarrow \forall x\, P(x)]$$

restringe il dominio dei numeri naturali a 0 e ai suoi successori.

Torniamo ora a considerare la definizione induttiva della chiusura transitiva di una relazione R: R^* è l'intersezione di tutti gli insiemi di coppie P tali che:

1. se $(a, b) \in R$ allora $(a, b) \in P$;

2. se $(a, b) \in R$ e $(b, c) \in P$, allora $(a, c) \in P$.

La definizione del corrispondente simbolo di predicato r^*, per rappresentare la chiusura transitiva di r, in logica del secondo ordine è la seguente:

$$\forall x \forall y \, (r^*(x,y) \equiv \quad \forall P \quad [\, \forall z \forall w \, (r(z,w) \rightarrow P(z,w)) \wedge \\ \forall z \forall v \forall w \, (r(z,v) \wedge P(v,w) \rightarrow P(z,w)) \\ \rightarrow P(x,y)])$$

La coppia (x, y) appartiene a R^* sse per ogni insieme P, se (a) P contiene tutte le coppie appartenenti a R e (b) contiene anche tutte le coppie (z, w) ogni volta che $(z, v) \in R$ e $(v, w) \in P$ per qualche v, allora (x, y) appartiene a P. Cioè (x, y) appartiene a R^* e e solo se (x, y) appartiene a ogni insieme P che soddisfa (a) e (b).

1.4 La negazione come fallimento in programmazione logica

La programmazione logica si basa sulla risoluzione SLD. Ma dal punto di vista logico da un insieme di clausole definite (programma) si possono derivare soltanto informazioni positive. Un goal è infatti sempre costituito da una congiunzione di atomi.

L'uso della negazione in Prolog allontana dunque il concetto di derivazione secondo il meccanismo di inferenza del Prolog dal concetto di inferenza logica da un insieme di clausole definite.

Ricordiamo che la semantica del Prolog si basa sull'*assunzione del mondo chiuso* (*closed world assumption*, CWA): ciò che non è una conseguenza logica della teoria è falso. Quest'assunzione è alla base del trattamento della negazione in Prolog; per derivare atomi negati, il Prolog utilizza la regola della *negazione come fallimento*: se il tentativo di dimostrare il goal A fallisce (in tempo finito), allora si può derivare $\neg A$.[1]

Per dare un fondamento logico alla negazione come fallimento occorre assumere che quando l'interprete Prolog tenta di dimostrare un atomo negato $\neg A$, A sia *ground*, cioè non contenga variabili che non sono istanziate al momento della chiamata. Infatti, in caso contrario il comportamento del Prolog può essere scorretto, come si può verificare con il semplice esempio seguente:

```
%% programma
p(a).
q(b).

%% esecuzione
?- \+ p(X), q(X).
false.
```

Il Prolog fallisce dunque quando tenta di dimotrare il goal $\neg p(X) \wedge q(X)$, sebbene $\neg p(b) \wedge q(b)$ sia derivabile (si tornerà in seguito su questo esempio).

Dato che la maggior parte degli interpreti Prolog non si comportano correttamente su atomi negati che contengono variabili non istanziate, deve essere cura del programmatore garantire che vengano valutati soltanto atomi negati *ground*, ad esempio mettendoli per ultimi nel corpo delle clausole.

[1] Nelle formule, utilizzeremo il simbolo logico $\neg$ anche per la negazione Prolog.

Per ricollegare la negazione come fallimento alla negazione logica occorre, a partire da un insieme P di clausole di programma (che possono contenere atomi negati nel corpo delle clausole), costruire un'estensione di P, chiamata il *completamento di Clark* (*Clark completion*) del programma.

Un programma *generale* (quale può essere ad esempio un programma Prolog) è un insieme di clausole della forma $A \leftarrow L_1 \wedge ... \wedge L_k$,[2] dove A è un atomo e $L_1, ..., L_k$ sono *letterali* (atomi o negazioni di un atomo).

Il primo passaggio della trasformazione del programma logico generale P nel suo completamento consiste nella riscrittura di ogni clausola in una forma equivalente. Sia C la clausola:

$$(C) \quad p(t_1, ..., t_n) \leftarrow L_1 \wedge ... \wedge L_k$$

La clausola C viene sostituita da

$$(C') \quad p(x_1, ..., x_n) \leftarrow x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge L_1 \wedge ... \wedge L_k$$

dove $x_1, ..., x_n$ sono variabili che non occorrono in C.

Sappiamo che, per convenzione, le variabili libere in una clausola sono da considerarsi quantificate universalmente; quindi la clausola C equivale a:

$$\forall y_1 ... \forall y_m \, (p(t_1, ..., t_n) \leftarrow L_1, ..., L_k)$$

dove $y_1, ..., y_m$ sono tutte le variabili che occorrono in C, e C' equivale a:

$$\forall x_1, ... \forall x_n \forall y_1 ... \forall y_m \, (p(x_1, ..., x_n) \leftarrow x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge L_1 \wedge ... \wedge L_k)$$

Quest'ultima formula equivale a sua volta a:

$$\forall x_1, ... \forall x_n \, (p(x_1, ..., x_n) \leftarrow \exists y_1 ... \exists y_m (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge L_1 \wedge ... \wedge L_k))$$

(dato che $y_1, ..., y_m$ non occorrono nel conseguente dell'implicazione $p(x_1, ..., x_n)$).

Infine, omettendo i quantificatori universali esterni secondo l'abituale convenzione, si ottiene la formula equivalente a C:

$$(C^*) \quad p(x_1, ..., x_n) \leftarrow \exists y_1 ... \exists y_m (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge L_1 \wedge ... \wedge L_k)$$

Quando questa trasformazione preliminare viene effettuate su tutte le clausole del programma logico, per ogni predicato p si ottiene un insieme di clausole della forma:

$$\begin{array}{c} p(x_1, ..., x_n) \leftarrow E_1 \\ \vdots \\ p(x_1, ..., x_n) \leftarrow E_q \end{array}$$

(la "procedura" che definisce p), dove ogni E_i ha la forma $\exists y_1 ... \exists y_m (x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge L_1 \wedge ... \wedge L_k)$.

Si consideri ora la formula:

$$\forall x_1, ... \forall x_n \, (p(x_1, ..., x_n) \leftarrow E_1 \vee ... \vee E_q)$$

Questa è equivalente all'insieme delle clausole che definiscono il predicato p.

[2] Utilizzeremo qui la notazione $A \leftarrow L_1 \wedge ... \wedge L_k$ anziché $L_1 \wedge ... \wedge L_k \rightarrow A$ o l'abituale notazione Prolog `A :- L1,...,Lk`.

Il *completamento* della definizione del predicato p, che indicheremo con $Comp(p)$, sostituisce l'implicazione con la doppia implicazione:

$$(Comp(p)) \quad \forall x_1, ...\forall x_n \, (p(x_1, ..., x_n) \equiv E_1 \vee ... \vee E_q)$$

Se un predicato p (a n argomenti) non occorre nella testa di alcuna clausola del programma P, il suo completamento è

$$\forall x_1, ...\forall x_n \, \neg p(x_1, ..., x_n)$$

Il completamento di un programma logico P, $Comp(P)$, è costituito dal completamento di tutti i predicati definiti in P.

È inoltre necessario aggiungere gli assiomi per l'uguaglianza e alcuni assiomi che chiameremo *assiomi del nome unico*:

1. $c \neq d$, per ogni coppia di costanti distinte c e d;
2. $\forall x_1 ...\forall x_n \, (f(x_1, ..., x_n) \neq c)$, per ogni simbolo funzionale f e costante c;
3. $\forall x_1 ...\forall x_n \, (f(x_1, ..., x_n) \neq g(x_1, ..., x_n))$, per ogni coppia di simboli funzionali distinti f e g;
4. $\forall x_1 ...\forall x_n \forall y_1 ...\forall y_n \, (x_1 \neq y_1 \vee ... \vee x_n \neq y_n \rightarrow f(x_1, ..., x_n) \neq f(y_1, ..., y_n))$ per ogni simbolo funzionale f;
5. per ogni termine $t[x]$ che contiene la variabile x e che non sia esso stesso una variabile: $\forall x \forall y_1 ...\forall y_n \, (t[x] \neq x)$, dove $x, y_1, ..., y_n$ sono tutte le variabili che occorrono in $t[x]$.

I primi tre gruppi di assiomi stabiliscono che due termini denotano lo stesso oggetto soltanto se il loro simbolo funzionale principale è lo stesso e gli argomenti sono identici. L'ultimo gruppo cattura l'effetto dell'*occur check* dell'algoritmo di unificazione.

Esempio 1 *Si consideri il programma P costituito dalle clausole:*

$$\begin{array}{l} europeo(peter) \\ europeo(x) \leftarrow italiano(x) \\ italiano(giorgio) \\ italiano(x) \leftarrow milanese(x) \\ milanese(silvio) \\ milanese(antonio) \end{array}$$

Il completamento di P è costituito dalle seguenti formule:

$$\begin{array}{ll} (Comp(europeo)) & \forall x \, (europeo(x) \equiv x = peter \vee italiano(x)) \\ (Comp(italiano)) & \forall x \, (italiano(x) \equiv x = giorgio \vee milanese(x)) \\ (Comp(milanese)) & \forall x \, (milanese(x) \equiv x = silvio \vee x = antonio) \\ peter \neq giorgio & \\ \vdots & \\ silvio \neq antonio & \end{array}$$

Si noti che $\neg europeo(carlo)$ non è una conseguenza logica di P, ma è una conseguenza logica di $Comp(P)$.

Il completamento di un programma logico generale P ha le seguenti proprietà:

1. P è una conseguenza logica di $Comp(P)$;
2. $Comp(P)$ non aggiunge informazioni positive: se A è una formula atomica, allora $P \models A$ se e solo se $Comp(P) \models A$.

Il seguente risultato è dovuto a Clark:

Teorema 1 (Teorema di Clark) *Sia P un programma logico generale e G un goal generale (che può contenere atomi negati). Si assuma inoltre che l'interprete Prolog si comporti correttamente sugli atomi negati (non tenti mai di dimostrare un goal negato in cui occorrono variabili non legate). Allora:*

1. *se il Prolog ha successo nel dimostrare il goal G a partire dal programma logico P fornendo la sostituzione di risposta θ, allora $Comp(P) \models \forall\,(G\theta)$.*[3]
2. *Se il Prolog termina con fallimento quando tenta di dimostrare il goal G a partire dal programma logico P, allora $Comp(P) \models \forall \neg G$.*

Si noti che nel punto 2 si dice "termina con fallimento", e non semplicemente che l'esecuzione non termina con successo (ad esempio perché non termina affatto).

Ad esempio, $\neg europeo(carlo)$ (che è un goal generale) è derivabile dal programma logico dell'esempio 1, ed è infatti una conseguenza logica del completamento del programma.

Torniamo ora all'ipotesi che l'interprete Prolog non deve mai dimostrare un atomo negato $\neg A$ con A non ground. Il completamento del programma $P = \{p(a), p(b)\}$ (già considerato a pagina 10) è:

$$\begin{array}{l}\forall x\,(p(x) \equiv x = a)\\ \forall x\,(q(x) \equiv x = b)\\ a \neq b\end{array}$$

Chiaramente, $Comp(P) \models \exists x\,(\neg p(x) \wedge q(x))$. Quindi, se l'interprete Prolog termina con fallimento sul goal $\neg p(X) \wedge q(X)$, il suo comportamento non è corretto rispetto alla semantica del completamento.

Come ultimo esempio, consideriamo il programma P costituito dalle clausole:

$$\begin{array}{l}genitore(a, a)\\ genitore(b, b)\\ antentato(x, y) \leftarrow genitore(x, y)\\ antentato(x, y) \leftarrow genitore(x, z), antenato(z, y)\end{array}$$

il cui completamento $Comp(P)$ è

$$\begin{array}{l}\forall x \forall y\,(genitore(x, y) \equiv (x = a \wedge y = a) \vee (x = b \wedge y = b))\\ \forall x \forall y\,(antenato(x, y) \equiv genitore(x, y) \vee \exists z\,(genitore(x, z) \wedge antenato(z, y)))\\ a \neq b\end{array}$$

[3] La notazione $\forall\,A$ indica la chiusura universale della formula A.

Si confronti $Comp(antentato)$ con la formula (1) dell'osservazione di pagina 7. Il teorema di Clark sembrerebbe contraddire il fatto che la chiusura transitiva di una relazione non è definibile in logica del primo ordine. Ma non è così. Come si è detto, $Comp(antentato)$ non definisce *antenato*, ma soltanto le sue "istanze positive". Ed infatti, il Prolog termina con successo quando tenta di dimostrare un goal della forma $antenato(c_1, c_2)$ con c_1 e c_2 costanti tali che $Comp(P) \models antenato(c_1, c_2)$. Ma se $Comp(P) \not\models antenato(c_1, c_2)$, la terminazione non è garantita. Si provi, ad esempio, ad eseguire il programma con goal $\neg\, antenato(a, b)$.

Capitolo 2

Il Calcolo delle Situazioni

2.1 Il linguaggio del Calcolo delle Situazioni

Il calcolo delle situazioni (*Situation Calculus*) fu inizialmente introdotto da John McCarthy nel 1963 [3, 4]. Quello che presentiamo in questo testo è basato sulla versione di Ray Reiter, proposta nel 1991 [5]. Una sua descrizione completa, che include anche il linguaggio Golog, si può trovare in [7].

Il calcolo delle situazioni è un formalismo logico progettato per rappresentare e ragionare su domini che cambiano dinamicamente. I cambiamenti nel mondo sono causati dall'esecuzione di *azioni*.

2.1.1 Situazioni e azioni

Secondo l'ontologia sottostante il calcolo delle situazioni, il mondo evolve attraverso una serie di *situazioni*, come risultato dell'esecuzione di varie azioni. Una situazione rappresenta una storia delle azioni eseguite. Nella versione di Reiter del calcolo delle situazioni (e contrariamente a quella originariamente definita da McCarthy e Hayes), una situazione non è uno stato, né un'istantanea del mondo, ma è una sequenza finita di azioni, una *storia*: una possibile evoluzione del mondo non è altro che una sequenza di azioni, che viene rappresentata da un termine detto *situazione*.

Sia le situazioni che le azioni sono rappesentate da termini (*reificazione* di situazioni e azioni, che sono cioè considerate come oggetti). Il linguaggio del calcolo delle situazioni è un linguaggio a molte sorte: ha infatti due tipi particolari per i termini che denotano situazioni e i termini che denotano azioni. Tutti gli altri oggetti del dominio sono rappresentati da termini di un terzo tipo (il tipo "oggetto"). Si possono usare (e quantificare) variabili di qualsiasi sorta. Quindi si può quantificare, in particolare, su situazioni e azioni.

Per le variabili di tipo azione utilizzeremo la lettera a, per quelle di tipo situazione la lettera s; x, y, z saranno usate per variabili di tipo oggetto (tutte le lettere qui indicate possono eventualmente essere corredate da apici o indici). Ad esempio, la formula $\forall x\, F$ significa che F vale per tutti gli elementi del dominio *di tipo oggetto*, e non dice nulla sulle azioni e sulle situazioni. Useremo le stesse lettere anche come *meta-variabili*, cioè per indicare un generico termine di tipo azione, situazione o oggetto; l'uso delle lettere come meta-variabili sarà comunque indicato sempre esplicitamente.

I termini di tipo azione possono essere costanti, oppure costruiti mediante l'applicazione di simboli funzionali ad altri termini. Ad esempio, si può utilizzare il termine di tipo azione $putOn(A, B)$ per rappresentare l'azione di mettere l'oggetto A sopra l'oggetto B ($putOn$ è un simbolo funzionale, A e B sono costanti di tipo oggetto). Analogamente, il termine $putOnTable(A)$ può denotare l'azione di mettere l'oggetto A sul tavolo.

La sequenza vuota di azioni, cioè la situazione esistente prima che venga eseguita qualunque azione, viene denotata da una costante, S_0, chiamata la *situazione iniziale*. Il linguaggio del calcolo delle situazioni utilizza inoltre un simbolo funzionale, *do*, per costruire situazioni: se a è un termine di tipo azione e s un termine di tipo situazione, allora il termine $do(a, s)$ indica la sequenza di azioni che consiste di quella denotata da s seguita dall'azione a. In altri termini, $do(a, s)$ è un termine di tipo situazione che denota la situazione che si ottiene eseguendo l'azione a nella situazione s.

Ad esempio, il termine di tipo situazione $do(putOnTable(B), S_0)$ denota la situazione che si ottiene a partire dalla situazione iniziale S_0 mettendo B sul tavolo. E $do(putOn(A, B), do(putOnTable(B), S_0))$ indica la situazione costituita, a partire dalla situazione iniziale, dalla sequenza di azioni "mettere B sul tavolo" e poi A sopra B.

Il fatto che le situazioni sono sequenze di azioni e non stati è evidenziato da uno degli assiomi generali del calcolo delle situazioni, che stabilisce che $do(a, s)$ è uguale a $do(a', s')$ se e soltanto se $a = a'$ e $s = s'$. Questa condizione non avrebbe senso se le situazioni fossero stati, perché due diverse azioni eseguite in due stati diversi potrebbero benissimo avere come risultato uno stesso stato.

Riassumendo:

1. Situazioni ed azioni sono considerate oggetti e rappresentate da termini (*reificazione* di situazioni e azioni);

2. I termini usati per rappresentare le azioni possono essere costanti o termini complessi, ad esempio: *forward, turn(right), turn(left).*

3. I termini usati per rappresentare le situazioni sono i seguenti:

   (a) La situazione iniziale (sequenza vuota di azioni) è rappresentata dalla costante S_0

   (b) Le altre situazioni sono rappresentate da termini della forma $do(a, s)$, dove *do* è un simbolo funzionale particolare, che si applica ad azioni e situazioni. Un termine della forma $do(a, s)$ è di tipo situazione: denota la situazione che risulta dall'esecuzione dell'azione a nella situazione s. Ad esempio: $do(forward, S_0)$ è di tipo situazione, come anche $do(turn(right), do(forward, S_0))$.

2.1.2 Fluenti

In generale, il valore di verità di un predicato o relazione può cambiare da una situazione ad un'altra. Predicati e relazioni non statiche (o *fluenti*), la cui verità, per dati argomenti, può variare nel tempo, sono rappresentati, nel calcolo delle situazioni, da simboli di predicato che hanno un argomento in più: un termine di tipo situazione come ultimo argomento. Chiameremo *fluenti* anche i simboli

di predicato di questo tipo, in opposizione ai simboli di predicato *statici* che non hanno parametri di tipo situazione.

Ad esempio, in un dominio in cui è possibile dipingere gli oggetti, si potrebbe usare il fluente *colore(x,y,s)* per indicare il fatto che il colore dell'oggetto x è y nella situazione s. Quindi, se $paint(x, y)$ indica l'azione di dipingere l'oggetto x del colore y si avrebbe:

$$\forall x \, \forall y \, \forall s \, colore(x, y, do(paint(x, y), s))$$

(il colore di x è y nella situazione che risulta da una qualsiasi situazione s colorando x del colore y).

Altri esempi di formule atomiche costruite mediante fluenti potrebbero essere i seguenti:

- $at(agent_1, stanza_2, S_0)$: l'agente 1 si trova nella stanza 2 nella situazione iniziale;
- $at(robot_1, stanza_3, do(forward(robot_1), S_0))$: dopo aver eseguito l'azione $forward(robot_1)$ nella situazione iniziale (il robot 1 va avanti di un passo), il robot 1 si trova nella stanza 3;
- $closed(door_1, do(lock(door_1), S_0))$: dopo aver eseguito l'azione di chiudere a chiave la porta 1 a partire dalla situazione iniziale, la porta 1 è chiusa.

Nel seguito utilizzeremo la parola "fluente" sia per indicare un simbolo di predicato che per denotare una formula atomica costruita utilizzando un fluente.

2.2 Rappresentazione del mondo nel calcolo delle situazioni

Per descrivere nel calcolo delle situazioni un dominio particolare, occorre innanzitutto determinare il linguaggio che si vuole utilizzare. Oltre ai simboli S_0 e *do* e al simbolo di uguaglianza (=), che sono comuni alla formalizzazione di ogni dominio nel calcolo delle situazioni, occorre definire:

1. l'insieme delle costanti di tipo oggetto e i simboli funzionali per la costruzione di termini di tipo oggetto;
2. l'insieme delle costanti di tipo azione e i simboli funzionali per la costruzione di termini di tipo azione;
3. l'insieme dei fluenti;
4. l'insieme (eventualmente vuoto) dei predicati statici.

In secondo luogo occorre definire una *teoria* nel Situation Calculus, cioè un insieme di assiomi, che descriva il mondo e le regole della sua evoluzione, utilizzando il linguaggio prescelto. Tale teoria deve includere:

1. assiomi che descrivano la situazione iniziale;
2. assiomi che descrivano le azioni, con le condizioni della loro applicabilità e i loro effetti.

Nel resto del capitolo descriveremo una metodologia generale che consente di assiomatizzare adeguatamente domini dinamici nel calcolo delle situazioni.

Considereremo come esempio il *mondo dei blocchi*, nel quale si ha un determinato numero di blocchi disposti su un tavolo, che possono formare torri. Considereremo in particolare il dominio in cui si hanno tre blocchi, A, B e C, inizialmente disposti come rappresentato nella figura 2.1.

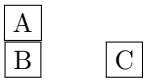

Figura 2.1: La situazione iniziale del mondo dei blocchi preso come esempio

È possibile spostare i blocchi sopra altri blocchi o sul tavolo, osservando le seguenti regole:

- si può spostare un solo blocco alla volta: un blocco x si può mettere sopra un blocco y o sul tavolo solo se sopra x non c'è alcun blocco;
- non si può mettere un blocco sopra se stesso;
- non si può mettere un blocco sopra un blocco già occupato: un blocco x si può mettere sopra un blocco y solo se sopra y non c'è alcun blocco;
- un blocco libero (senza alcun blocco sopra di esso) si può sempre spostare sul tavolo (sul tavolo c'è sempre spazio a sufficienza).

Per rappresentare il mondo dei blocchi considerato come esempio utilizzeremo il linguaggio seguente:

1. le costanti A, B e C, di tipo oggetto;
2. i seguenti simboli funzionali per la costruzione di azioni:

   $$putOn : oggetto \times oggetto \rightarrow azione$$
   $$putOnTable : oggetto \rightarrow azione$$

   La notazione utilizzata sopra determina il *tipo* di ciascun simbolo funzionale, cioè il numero e il tipo degli argomenti e il tipo del termine costruito applicando il simbolo funzionale ai termini appropriati (in questo caso, sempre "azione"). Dunque se x e y sono termini di tipo oggetto, $putOn(x, y)$ e $putOnTable(x)$ sono termini di tipo azione.
3. i seguenti fluenti:

   $$on : oggetto \times oggetto \times situazione$$
   $$onTable : oggetto \times situazione$$

   Analogamente alla notazione utilizzata per le azioni, il tipo di un fluente è espresso come il prodotto cartesiano dei tipi dei suoi argomenti. Dunque, se x e y sono termini di tipo oggetto e s un termine di tipo situazione, le espressioni $on(x, y, s)$ e $onTable(x, s)$ sono formule atomiche del linguaggio.
4. nessun predicato statico.

2.2.1 Rappresentazione della situazione iniziale

La situazione iniziale è rappresentata da un insieme di formule che descrivono in maniera completa tutto ciò che è vero e tutto ciò che è falso inizialmente. Nel nostro esempio possiamo rappresentare la situazione iniziale mediante il seguente insieme di assiomi:

$$\begin{array}{lll} \{ \quad on(A,B,S_0), & onTable(B,S_0), & onTable(C,S_0), \\ \quad \neg\, on(A,C,S_0), & \neg\, on(B,A,S_0), & \neg\, on(B,C,S_0), \\ \quad \neg\, on(C,A,S_0), & \neg\, on(C,B,S_0), & \neg\, onTable(A,S_0), \\ \quad \forall x \neg\, on(x,x,S_0) & \} & \end{array}$$

Come si vede è necessario rappresentare sia le informazioni "positive" (cioò che è vero), sia quelle "negative" (ciò che è falso).

Un modo alternativo, più compatto, di rappresentare lo stato iniziale dell'esempio del mondo dei blocchi considerato è il seguente:

$$\begin{array}{ll} \{ \quad \forall x \forall y \, (on(x,y,S_0) \equiv x = A \wedge y = B) \\ \quad \forall x \, (onTable(x,S_0) \equiv x = B \vee x = C) \quad \} \end{array}$$

Questo insieme di formule è logicamente equivalente al precedente se si assume anche un insieme di *assiomi del nome unico* per gli oggetti:

$$\{A \neq B, \quad A \neq C, \quad B \neq C \quad \}$$

Gli assiomi che descrivono lo stato iniziale vengono chiamati *initial state axioms*.

Un altro esempio può essere il seguente: si consideri un mondo in cui si ha un unico agente, che si trova, in ogni situazione, in un determinato luogo e possiede determinati oggetti; inizialmente si trova a casa e ha un litro di latte. Gli assiomi dello stato iniziale possono essere i seguenti:

$$\begin{array}{ll} \{ \quad \forall x \, (at(x,S_0) \equiv x = home) \\ \quad \forall x \, (have(x,S_0 \equiv x = milk) \quad \} \end{array}$$

(assieme agli appropriati assiomi del nome unico per gli oggetti).

2.2.2 Rappresentazione delle azioni

Ogni azione a ha due caratteristiche fondamentali che devono essere adeguatamente rappresentate: delle *precondizioni*, cioè le condizioni che devono essere soddisfatte in una situazione s perché a possa essere eseguita in s; e degli *effetti*, cioè i cambiamenti provocati dall'esecuzione di a.

Considereremo dapprima un approccio semplice per la rappresentazione delle azioni, per poi raffinarlo in modo da ottenere un'assiomatizzazione più efficace.

Per ogni azione a, con parametri $x_1, ..., x_n$, si determina:

1. Una formula $precond_a(x_1, ..., x_n, s)$ che esprime le condizioni di applicabilità di $a(x_1, ..., x_n)$ nella situazione s; $precond_a(x_1, ..., x_n, s)$ rappresenta cioè il fatto che $a(x_1, ..., x_n)$ è eseguibile nella situazione s.

2. Una formula $effect_a(x_1, ..., x_n, s)$ che esprime l'effetto dell'esecuzione di $a(x_1, ..., x_n)$ nella situazione s.

e si include nella teoria la descrizione dell'effetto di $a(x_1, ..., x_n)$:

$$\forall s \, \forall x_1 ... \forall x_n \, (precond_a(x_1, ..., x_n, s) \rightarrow effect_a(x_1, ..., x_n, do(a(x_1, ..., x_n), s)))$$

Questo schema di formula rappresenta il fatto che se, in una qualsiasi situazione s sono vere le precondizioni per l'eseguibilità di $a(x_1, ..., x_n)$, allora gli effetti di tale azione valgono nella situazione $do(a(x_1, ..., x_n), s)$.

Si noti che le notazioni $precond_a(x_1, ..., x_n, s)$ e $effect_a(x_1, ..., x_n, s)$ non sono formule, ma *meta*-formule, cioè abbreviazioni che rappresentano formule del calcolo delle situazioni che possono contenere le variabili libere $x_1, ..., x_n, s$ (e non altre).

Ad esempio, nel caso del mondo dei blocchi, le precondizioni per le azioni $putOnTable(x)$ e $putOn(x, y)$ possono essere espresse, rispettivamente, dalle formule:

$$\begin{array}{lll} precond_{putOnTable}(x, s) & \equiv_{def} & \neg onTable(x, s) \wedge \neg \exists y \, on(y, x, s) \\ precond_{putOn}(x, y, s) & \equiv_{def} & \neg on(x, y, s) \wedge \neg (x = y) \wedge \neg \exists z \, on(z, x, s) \\ & & \wedge \neg \exists z \, on(z, y, s) \end{array}$$

(ricordiamo che utilizziamo $x \neq y$ come abbreviazione della formula $\neg(x = y)$).

Gli effetti delle due azioni possono essere rappresentati dalle formule seguenti:

$$\begin{array}{lll} effect_{putOnTable}(x, s) & \equiv_{def} & onTable(x, do(putOnTable(x), s)) \wedge \\ & & \neg \exists y \, on(x, y, do(putOnTable(x), s)) \\ effect_{putOn}(x, y, s) & \equiv_{def} & on(x, y, do(putOn(x, y), s)) \wedge \\ & & \neg \, onTable(x, do(putOn(x, y), s)) \wedge \\ & & \forall z \, (z \neq y \rightarrow \neg \, on(x, z, do(putOn(x, y), s))) \end{array}$$

Essi stabiliscono che l'effetto di $putOnTable(x)$ è quello di avere x sul tavolo e non più sopra un (qualsiasi) blocco; e gli effetti di $putOn(x, y)$ sono che x è sopra y, e non è più sul tavolo né sopra altri blocchi diversi da y.

La teoria che rappresenta il mondo dei blocchi includerà allora i seguenti assiomi:

$$\begin{array}{l} \forall s \, \forall x \, (\neg onTable(x, s) \wedge \neg \exists y \, on(y, x, s) \rightarrow \\ \quad onTable(x, do(putOnTable(x), s)) \wedge \neg \exists y \, on(x, y, do(putOnTable(x), s))) \\ \forall s \, \forall x \, \forall y \, (\neg on(x, y, s) \wedge x \neq y \wedge \neg \exists z \, on(z, x, s) \wedge \neg \exists z \, on(z, y, s) \rightarrow \\ \quad on(x, y, do(putOn(x, y), s)) \wedge \neg \, onTable(x, do(putOn(x, y), s)) \wedge \\ \quad \forall z \, (z \neq y \rightarrow \neg \, on(x, z, do(putOn(x, y), s)))) \end{array}$$

Ma questi assiomi, chiamati *effect axioms*, non sono sufficienti: essi descrivono solo cosa cambia quando viene eseguita un'azione. È necessario anche descrivere *cosa non cambia*. Per ogni azione, si deve descrivere tutto quello che non cambia quando si esegue l'azione, mediante quelli che sono chiamati *frame axioms*.

Nel nostro esempio, la teoria dovrebbe includere anche tutti gli assiomi necessari per rappresentare il fatto che, quando si mette x sul tavolo, la verità o falsità di $onTable(x', s)$ e $on(x', y, s)$, per $x' \neq x$, non cambia. Per costruire un tale insieme di assiomi, possiamo iniziare a determinare, per ogni fluente, quali sono le azioni che *non* ne modificano la verità o falsità, assieme alle condizioni sotto le quali essi restano veri o falsi. Nel nostro caso, queste informazioni sono riassunte nella tabella 2.1. Nella tabella, ad ogni fluente sono riservate due

fluente	*azione*	*condizione*
$on(x, y, s)$	$a = putOn(x', y')$	se $x \neq x'$
	$a = putOnTable(x')$	se $x \neq x'$
$\neg on(x, y, s)$	$a = putOn(x', y')$	se $x \neq x'$ o $y \neq y'$
	$a = putOnTable(x')$	
$onTable(x, s)$	$a = putOn(x', y)$	se $x \neq x'$
	$a = putOnTable(x')$	
$\neg onTable(x, s)$	$a = putOn(x', y')$	
	$a = putOnTable(x')$	se $x \neq x'$

Tabella 2.1: Tabella con le informazioni per costruire i frame axioms

insiemi di righe: il primo insieme considera le azioni che non modificano la sua verità, il secondo quelle che non cambiano la sua falsità.

Ogni riga della tabella corrisponderà allora a un *frame axiom*:

$$\forall s \forall x \forall y \forall x' \forall y' (on(x, y, s) \wedge precond_{putOn}(x', y', s) \wedge x \neq x' \rightarrow on(x, y, do(putOn(x', y'), s)))$$
$$\forall s \forall x \forall y \forall x' (on(x, y, s) \wedge precond_{putOnTable}(x', s) \wedge x \neq x' \rightarrow on(x, y, do(putOnTable(x', y'), s)))$$
$$\forall s \forall x \forall y \forall x' \forall y' (\neg on(x, y, s) \wedge precond_{putOn}(x', y', s) \wedge (x \neq x' \vee y \neq y') \rightarrow on(x, y, do(putOn(x', y'), s)))$$
$$\forall s \forall x \forall y \forall x' (\neg on(x, y, s) \wedge precond_{putOnTable}(x', s) \rightarrow \neg on(x, y, do(putOnTable(x', y'), s)))$$
$$\forall s \forall x \forall x' \forall y' (onTable(x, s) \wedge precond_{putOn}(x', y', s) \wedge x \neq x' \rightarrow onTable(x, do(putOn(x', y'), s)))$$
$$\forall s \forall x \forall x' (onTable(x, s) \wedge precond_{putOnTable}(x', s) \rightarrow onTable(x, do(putOnTable(x'), s)))$$
$$\forall s \forall x \forall x' \forall y' (\neg onTable(x, s) \wedge precond_{putOn}(x', y', s) \rightarrow \neg onTable(x, do(putOn(x', y'), s)))$$
$$\forall s \forall x \forall x' (\neg onTable(x, s) \wedge precond_{putOnTable}(x', s) \wedge x \neq x' \rightarrow \neg onTable(x, do(putOnTable(x'), s)))$$

Il primo di essi, ad esempio, stabilisce che se in una qualsiasi situazione s, il blocco x è sopra il blocco y e se valgono le precondizioni per eseguire $putOn(x', y')$ per $x' \neq x$, allora l'esecuzione di tale azione nella situazione s non modifica la verità del fatto che x sia sopra y.

Il numero di tali assiomi è normalmente molto elevato e questo costituisce quello che viene chiamato "problema del frame" (*frame problem*). Normalmente, infatti, i fluenti modificati da un'azione sono pochi e molti invece quelli non modificati; per cui il numero di frame axioms è normalmente molto vicino a $2 \times \mathcal{F} \times \mathcal{A}$, dove $\mathcal{F}$ è il numero di fluenti, e $\mathcal{A}$ il numero di azioni del dominio. Infatti, per ogni fluente e ogni azione che non lo modifica si avrà un frame axiom positivo e uno negativo (cioè un assioma che stabilisce che il fluente resta vero quando si esegue l'azione, e uno che stabilisce che esso resta falso). Se si considera un dominio con 100 azioni e 100 fluenti, il numero di frame axioms sarebbe vicino a 20.000. Il grande numero di frame axioms rappresenta un problema sia per chi

deve assiomatizzare il dominio, sia per una sua implementazione, che dovrebbe ragionare in modo efficiente in presenza di così tanti assiomi.

La descrizione degli effetti delle azioni può essere resa più compatta sfruttando la possibilità di quantificare sulle azioni. Il modo di rappresentare le azioni descritto nel seguito fu proposto da R. Reiter [5].

Essenzialmente, viene separata la descrizione delle precondizioni delle azioni da quella dei loro effetti, e, per la seconda, anziché focalizzarsi sulle azioni ci si focalizza sui fluenti.

Assiomi delle precondizioni

Viene innanzitutto introdotto un nuovo predicato, $Poss$, che si applica ad un'azione e una situazione: $Poss(a, s)$ rappresenta il fatto che è possibile eseguire l'azione a nella situazione s.

Per ogni azione a, con parametri $x_1, ..., x_n$, la teoria contiene un *assioma delle precondizioni* (*precondition axiom*), che ha la forma seguente:

$$\forall s \forall x_1 ... \forall x_n (precond_a(x_1, ..., x_n, s) \rightarrow Poss(a(x_1, ..., x_n), s))$$

Ad esempio, nel mondo dei blocchi avremo due assiomi delle precondizioni:

$$\forall s \forall x (\neg onTable(x, s) \wedge \neg \exists y\, on(y, x, s) \rightarrow Poss(putOnTable(x), s))$$
$$\forall s \forall x \forall y (\neg\, on(x, y, s) \wedge x \neq y \wedge \neg \exists z\, on(z, x, s) \wedge \neg \exists z\, on(z, y, s)) \rightarrow Poss(PutOn(x, y), s))$$

Assiomi dello stato successore

Per descrivere gli effetti delle azioni si sfrutta la possibilità di quantificare sulle azioni stesse e ci si focalizza sui fluenti: per ogni fluente $R(x_1, ..., x_n, s)$, si determinano:

1. quali sono le azioni che fanno diventare vero $R(x_1, ..., x_n)$, con eventuali condizioni aggiuntive;

2. quali sono le azioni che fanno diventare falso $R(x_1, ..., x_n)$, con eventuali condizioni aggiuntive.

Conseguentemente si determinano due formule:

$G_R^+(a, s, x_1, ..., x_n)$**,** che rappresenta tutte le condizioni che possono causare il "passaggio" di $R(x_1, ..., x_n)$ da falso a vero;

$G_R^-(a, s, x_1, ..., x_n)$**,** che rappresenta tutte le condizioni che possono causare il "passaggio" di $R(x_1, ..., x_n)$ da vero a falso.

(si noti che, anche qui, $G_R^+(a, s, x_1, ..., x_n)$ e $G_R^-(a, s, x_1, ..., x_n)$ non sono formule, ma abbreviazioni per formule del calcolo delle situazioni, in cui possono occorrere libere soltanto le variabli $a, s, x_1, ..., x_n$). Normalmente, G_R^+ e G_R^- contengono la variabile a, e possono contenere anche la variabile s; infatti tali formule esprimeranno condizioni del tipo: "a è una delle azioni ....", eventualmente in congiunzione con condizioni che devono valere nello stato s.

Ad esempio, si consideri un mondo in cui gli oggetti si possono rompere in due modi: facendoli cadere, se sono fragili, oppure facendo esplodere una bomba

ad essi vicina. La formula $G^{+}_{broken}(a, s, x)$ per il fluente *broken*, potrebbe essere la seguente:

$$(a = drop(x) \wedge fragile(x)) \vee \exists y\,(a = explode(y) \wedge near(x, y, s))$$

(a è l'azione di far cadere x e x è fragile – predicato statico – oppure a è l'azione di far esplodere una qualche bomba y che si trova vicino a x nella situazione s).

Per ogni fluente $R(x_1, ..., x_n, s)$, la teoria includerà allora l'*assioma di stato successore* per R, che ha la forma seguente:

$$\begin{aligned}\forall a \forall s \forall x_1 ... \forall x_n \{ Poss(a, s) \;\rightarrow\; & \\ & [R(x_1, ..., x_n, do(a, s)) \equiv \\ & \quad G^{+}_{R}(a, s, x_1, ..., x_n) \vee \\ & \quad (R(x_1, ..., x_n, s) \wedge \neg G^{-}_{R}(a, s, x_1, ..., x_n))]\}\end{aligned}$$

Esso si può leggere come segue: per ogni azione a, per ogni situazione s e per tutti gli oggetti $x_1, ..., x_n$: la relazione R vale tra gli oggetti $x_1, ..., x_n$ nella situazione $do(a, s)$ se e solo se a è una delle azioni che fanno diventare vero R, per $x_1, ..., x_n$, R vale per $x_1, ..., x_n$ in s e a non lo modifica.

Nell'esempio sopra considerato, se vi è anche la possibilità di riparare oggetti (azione che fa diventare falso *broken*), l'assioma dello stato successore per il fluente *broken* è il seguente:

$$\begin{aligned}\forall a \forall s \forall y \{ Poss(a, s) \rightarrow [broken(x, do(a, s)) \equiv & \\ ((a = drop(x) \wedge fragile(x)) \vee \exists y\,(a = explode(y) \wedge near(x, y, s)) & \\ \vee (broken(x, s) \wedge \neg(a = repair(x))))]\}\end{aligned}$$

che si può leggere: "per ogni azione a, per ogni situazione s e per ogni oggetto y: x è rotto nella situazione $do(a, s)$ se e solo se a è un'azione che rompe x (cioè con a si fa cadere x che è fragile, oppure si fa esplodere una bomba vicino a x), oppure x è rotto in s e l'azione a non è quella di ripararlo.

Si noti che il numero di assiomi dello stato successore per un determinato dominio è uguale al numero di fluenti.

Consideriamo ora l'esempio del mondo dei blocchi. Determiniamo innanzitutto le azioni che possono far diventare vero o falso ciascun fluente, come rappresentato nella tabella 2.2.

fluente	*azione*	*condizione*
$on(x, y, s)$	$a = putOn(x, y)$	
$\neg on(x, y, s)$	$a = putOn(x, y')$ $a = putOnTable(x)$	se $y \neq y'$
$onTable(x, s)$	$a = putOnTable(x)$	
$\neg onTable(x, s)$	$a = putOn(x, y)$	

Tabella 2.2: Tabella con le informazioni per costruire gli assiomi dello stato successore nel mondo dei blocchi

Le formule G^+_R, G^-_R per i due fluenti ($R = on$ e $R = onTable$) sono le seguenti:

$$\begin{array}{lll} G^+_{on}(a,s,x,y) & \equiv_{def} & a = putOn(x,y) \\ G^-_{on}(a,s,x,y) & \equiv_{def} & \exists y' \, (a = putOn(x,y') \wedge y \neq y') \vee a = putOnTable(x) \\ G^+_{onTable}(a,s,x) & \equiv_{def} & a = putOnTable(x) \\ G^-_{onTable}(a,s,x) & \equiv_{def} & \exists y \; a = putOn(x,y) \end{array}$$

Si noti che in realtà la condizione $y \neq y'$ potrebbe essere omessa in $G^-_{on}(a,s,x,y)$, in quanto se $y = y'$ non è possibile eseguire $putOn(x,y')$ in una situazione in cui x è già sopra y. Si osservi inoltre che le uniche variabili libere nelle formule sopra elencate sono quelle indicate tra parentesi, come "pseudo-argomenti" di G^+_R e G^-_R.

I due assiomi dello stato successore sono dunque i seguenti:

$$\begin{array}{l} \forall a \forall s \forall x \forall y \{ Poss(a,s) \rightarrow \\ \qquad [on(x,y,do(a,s)) \equiv \\ \qquad\quad a = putOn(x,y) \\ \qquad\quad \vee (on(x,y,s) \wedge \neg (\exists y' \, (a = putOn(x,y') \wedge y \neq y') \\ \qquad\qquad\qquad\qquad \vee a = putOnTable(x))]\} \end{array}$$

$$\begin{array}{l} \forall a \forall s \forall x \{ Poss(a,s) \rightarrow \\ \qquad [onTable(x,do(a,s)) \equiv \\ \qquad\quad a = putOnTable(x) \\ \qquad\quad \vee (onTable(x,s) \wedge \neg \exists y \; a = putOn(x,y))]\} \end{array}$$

A proposito dell'assioma dello stato successore per on, osserviamo che la formula $\neg(\exists y' \, (a = putOn(x,y') \wedge y \neq y') \vee a = putOnTable(x))$ è equivalente a: $\neg \exists y' \, (a = putOn(x,y') \wedge y \neq y') \wedge \neg a = putOnTable(x)$. In altri termini, perché la verità di $on(x,y,s)$ non cambi l'azione a deve essere diversa sia da $putOn(x,y')$ con $y \neq y'$, sia da $putOnTable(x)$.

Si osservino inoltre due cose importanti:

1. gli assiomi dello stato successore non menzionano le condizioni di eseguibilità delle azioni. Di esse tiene conto l'antecedente dell'implicazione, $Poss(a,s)$. Le uniche condizioni "di contorno" che vanno esplicitate negli assiomi dello stato successore sono quelle che possono determinare effetti diversi di un'azione sul fluente (ad esempio, far cadere un oggetto non fragile non lo rompe; mettere x su y' non rende falso $on(x,y,s)$ se $y = y'$).

2. Quando una delle condizioni G^+_R o G^-_R devono far riferimento a azioni che hanno parametri diversi dagli argomenti di R, tali variabili non possono essere lasciate libere né quantificate universalmente all'esterno dell'assioma dello stato successore. Esse vanno invece quantificate esistenzialmente all'interno delle condizioni stesse.

Come esempio della seconda osservazione, consideriamo un dominio in cui alcuni robot possono spostarsi da un posto a un altro. L'unica azione è $go(x,y,z)$ (il robot x va da x a z) e l'unico fluente è $at(x,y,s)$ (il robot x si trova nel posto y nella situazione s). L'assioma dello stato successore per il fluente at è il

seguente:

$$\begin{aligned}\forall a \forall s \forall x \{ Poss(a,s) \rightarrow \\ [at(x,y,do(a,s)) \equiv \\ \exists z\, a = go(x,z,y) \\ \vee (at(x,y,s) \wedge \neg \exists z\ a = go(x,y,z))]\}\end{aligned}$$

Infine, osserviamo che da un assioma dello stato successore, della forma

$$\begin{aligned}\forall a \forall s \forall x_1 ... \forall x_n \{ Poss(a,s)\ \rightarrow \\ [R(x_1,...,x_n,do(a,s)) \equiv \\ G_R^+(a,s,x_1,...,x_n) \vee \\ (R(x_1,...,x_n,s) \wedge \neg G_R^-(a,s,x_1,...,x_n))]\}\end{aligned}$$

sono derivabili le due formule seguenti:

$$\begin{aligned}&1) \quad \forall a \forall s \forall x_1 ... \forall x_n (Poss(a,s) \wedge G_R^+(a,s,x_1,...,x_n) \\ &\qquad\qquad \rightarrow R(x_1,...,x_n,do(a,s))) \\ &2) \quad \forall a \forall s \forall x_1 ... \forall x_n (Poss(a,s) \wedge R(x_1,...,x_n,s) \wedge \neg G_R^-(a,s,x_1,...,x_n) \\ &\qquad\qquad \rightarrow R(x_1,...,x_n,do(a,s)))\end{aligned}$$

La formula 1 rappresenta proprio gli assiomi degli effetti per tutte le azioni. E la 2 gli assiomi del frame.

Come esempi, sono conseguenze logiche dell'assioma dello stato successore per il fluente *OnTable*, nella teoria del mondo dei blocchi, le formule:

$$\forall a \forall s (Poss(a,s) \wedge a = putOnTable(A) \rightarrow onTable(A, do(a,s)))$$

quindi anche

$$\begin{aligned}&\forall s\, (Poss(putOnTable(A),s) \rightarrow onTable(A, do(putOnTable(A),s))) \\ &\forall a \forall s (Poss(a,s) \wedge onTable(A,s) \wedge \neg \exists y\ a = putOn(A,y) \rightarrow onTable(A, do(a,s)))\end{aligned}$$

(subito dopo aver messo A sul tavolo, A è sul tavolo; e A resta sul tavolo se non si mette sopra un altro blocco).

2.2.3 Altri assiomi

Se il mondo descritto ha predicati statici, la corrispondente teoria del calcolo delle situazioni dovrà contenere degli assiomi appropriati che li definiscano. Ad esempio, in un dominio in cui sono coinvolti diversi tipi di oggetti (come robot, posti, oggetti trasportabili, ecc.) e vogliamo distinguerne il tipo, dobbiamo utilizzare dei predicati statici ed i corrispondenti assiomi:

$$\begin{aligned}&\forall x\, (robot(x) \equiv x = r_1 \vee x = r_2) \\ &\forall x\, (posto(x) \equiv x = p_1 \vee x = p_2 \vee x = p_3)\end{aligned}$$

E devono essere in ogni caso descritte tutte le proprietà dei predicati statici. Ad esempio:

$$\begin{aligned}&\forall x\, (supermarket(x) \rightarrow sells(x, milk)) \\ &\forall x \forall y \forall z (door(x) \wedge connect(x,y,z) \rightarrow connect(x,z,y))\end{aligned}$$

Inoltre, per molti scopi avremo bisogno di *assiomi del nome unico* per gli oggetti (già introdotti nel paragrafo 2.2.1). Nell'esempio del mondo dei blocchi:

$$\{ A \neq B, \quad A \neq C, \quad B \neq C \quad \}$$

Oltre a questi, dobbiamo assiomatizzare il fatto che gli oggetti del dominio sono *soltanto* quelli menzionati. Nel nostro caso:

$$\forall x\,(x = A \vee x = B \vee x = C)$$

(ricordiamo che $\forall x$ è una quantificazione sugli elementi di tipo oggetto). Senza questo assioma, che chiameremo assioma del dominio finito, non si potrebbe ad esempio derivare:

$$\neg\exists x\, on(x, C, S_0)$$

dagli assiomi dello stato iniziale per il nostro esempio.

Ogni teoria nel calcolo delle situazioni avrà inoltre altri assiomi di carattere generale: gli *assiomi del nome unico* per le azioni (*unique name axioms*), che stabiliscono che azioni con nomi diversi sono diverse e azioni con argomenti diversi sono diverse.

Nel caso del mondo dei blocchi, si avranno ad esempio:

$$\begin{array}{ll} a) & \forall x\forall y\,(putOn(x,y) \neq putOnTable(x)) \\ b) & \forall x\forall y\forall x'\forall y'\,(putOn(x,y) = putOn(x',y') \rightarrow x = x' \wedge y = y') \\ c) & \forall x\forall y\,(putOnTable(x) = putOnTable(y) \rightarrow x = y) \end{array}$$

Tali assiomi sono necessari per poter utilizzare, negli assiomi dello stato successore, la parte $\neg G_R^-$, cioè la formula 2 a pagina 25. Ad esempio, nella teoria del mondo dei blocchi per il caso considerato si utilizzano gli assiomi del nome unico per derivare $onTable(C, do(putOn(A, C), S_0)$. Senza gli assiomi del nome unico per le azioni, è possibile infatti derivare

$$(\alpha)\quad \neg\exists y\ PutOn(A, C) = putOn(C, y) \rightarrow onTable(C, do(PutOn(A, C), S_0))$$

(si veda la figura 2.2 – dove sono state usate molte regole derivate del calcolo di deduzione naturale). Ma per derivare $onTable(C, do(PutOn(A, C), S_0))$ da (α) è necessario utilizzare gli assiomi del nome unico: si assumano infatti

$$\begin{array}{lll} 1) & \exists y\ PutOn(A, C) = putOn(C, y) & \text{ipotesi} \\ 2) & PutOn(A, C) = putOn(C, y) & \text{ipotesi} \end{array}$$

(1 è assunta allo scopo di ridurla all'assurdo, 2 allo scopo di applicare la regola di eliminazione del quantificatore esistenziale). Da 2 si può derivare $A = C$ dall'assioma del nome unico (a). Ora, dall'assioma del nome unico per gli oggetti $A \neq C$ si deriva la contraddizione, $\perp$. Quindi, $\perp$ è derivabile da 1 (scaricando l'ipotesi 2), e quindi abbiamo (introduzione di $\neg$): $\neg\exists y\ PutOn(A, C) = putOn(C, y)$. Da quest'ultima formula e α possiamo infine dedurre che C è ancora sul tavolo quando si mette A sopra di esso: $onTable(C, do(PutOn(A, C), S_0))$.

Infine, le teorie nel calcolo delle situazioni avranno alcuni assiomi di carattere fondazionale per il calcolo delle situazioni, indipendenti dal dominio: gli assiomi del nome unico per le situazioni:

$$\begin{array}{l} \forall a\,\forall s\,S_0 \neq do(a, s) \\ \forall a\,\forall s\,\forall a'\,\forall s'\,(do(a, s) = do(a', s') \rightarrow a = a' \wedge s = s') \end{array}$$

e un assioma di induzione [6], che richiede la logica del secondo ordine:

$$\forall P\,[P(S_0) \wedge \forall a\forall s(P(s) \rightarrow P(do(a, s))) \rightarrow \forall s\,P(s)]$$

(1)	$onTable(C, S_0)$	initial state axiom
(2)	$\neg\, on(A, C, S_0)$	initial state axiom
(3)	$A \neq C$	unique name axiom for objects
(4)	$\neg\exists z\, on(z, C, S_0)$	derivabile dagli initial state axioms e dall'assioma del dominio finito
(5)	$\neg\exists z\, on(z, A, S_0)$	derivabile dagli initial state axioms e dall'assioma del dominio finito
(6)	$\forall s\forall x\forall y(\neg on(x, y, s) \wedge x \neq y \wedge \neg\exists z\, on(z, x, s) \wedge \neg\exists z\, on(z, y, s)) \rightarrow Poss(PutOn(x, y), s))$	precondition axiom
(7)	$\neg on(A, C, S_0) \wedge A \neq C \wedge \neg\exists z\, on(z, C, S_0) \wedge \neg\exists z\, on(z, A, S_0)) \rightarrow Poss(PutOn(A, C), S_0)$	da 6, per istanziazione
(8)	$Poss(PutOn(A, C), S_0)$	da 2,3,4,5,7
(9)	$\forall a\forall s\forall x\{Poss(a, s) \rightarrow [onTable(x, do(a, s)) \equiv a = putOnTable(x) \vee (onTable(x, s) \wedge \neg\exists y\ a = putOn(x, y))]\}$	successor state axiom
(10)	$Poss(PutOn(A, C), S_0) \rightarrow [onTable(C, do(PutOn(A, C), S_0)) \equiv PutOn(A, C) = putOnTable(C) \vee (onTable(C, S_0) \wedge \neg\exists y\ PutOn(A, C) = putOn(C, y))]$	da 9, per istanziazione
(11)	$onTable(C, do(PutOn(A, C), S_0)) \equiv PutOn(A, C) = putOnTable(C) \vee (onTable(C, S_0) \wedge \neg\exists y\ PutOn(A, C) = putOn(C, y))$	da 8 e 10
(12)	$onTable(C, S_0) \wedge \neg\exists y\ PutOn(A, C) = putOn(C, y)) \rightarrow onTable(C, do(PutOn(A, C), S_0)$	da 11
(13)	$\neg\exists y\ PutOn(A, C) = putOn(C, y) \rightarrow onTable(C, do(PutOn(A, C), S_0))$	da 1 e 12

Figura 2.2: Una derivazione nella teoria del mondo dei blocchi

Proprio come l'assioma di induzione per i numeri naturali restringe il dominio dei numeri a 0 e i suoi successori, così l'effetto dell'assioma di induzione sulle situazioni è quello di restringere il dominio delle situazioni: in qualsiasi modello l'insieme delle situazioni sarà costituito dal più piccolo insieme $\mathcal{C}$ che soddisfa le condizioni seguenti:

- $S_0 \in \mathcal{C}$;
- se $S \in \mathcal{C}$ e a è un'azione, allora $do(a, S) \in \mathcal{C}$.

In altre parole, non vi sono altre situazioni se non quelle raggiungibili dalla situazione iniziale eseguendo una sequenza di azioni.

2.3 Pianificazione deduttiva nel calcolo delle situazioni

Storicamente, il calcolo delle situazioni è stato spesso identificato con le applicazioni della pianificazione in intelligenza artificiale. Un problema di pianificazione consiste in questo: data una descrizione dello stato iniziale e di un obiettivo da raggiungere, trovare una sequenza di azioni che, se eseguite, portano dalla situazione iniziale a una situazione in cui è vero l'obiettivo.

Come esempio, consideriamo il caso del mondo dei blocchi con la situazione iniziale assiomatizzata nel paragrafo 2.2.1. Supponiamo di avere il semplice problema di pianificazione con l'obiettivo di avere il blocco A sul tavolo. Come si è visto a pagina 25, dalla teoria del mondo dei blocchi si può derivare

$$\forall s\,(Poss(putOnTable(A), s) \rightarrow onTable(A, do(putOnTable(A), s)))$$
quindi anche
$$Poss(putOnTable(A), S_0) \rightarrow onTable(A, do(putOnTable(A), S_0))$$

Poiché dagli assiomi della situazione iniziale si può derivare

$$\neg onTable(A, S_0) \wedge \neg\exists y\, on(y, A, S_0)$$

dall'assioma delle precondizioni per $putOnTable$ si deriva $Poss(putOnTable(A), S_0)$, dunque nella teoria dei blocchi si deriva

$$onTable(A, do(putOnTable(A), S_0))$$
e, per introduzione di $\exists$
$$\exists s\, onTable(A, s)$$

Dunque esiste una sequenza di azioni (in questo caso costituita soltanto dall'azione $putOnTable(A)$) che porta dalla situazione iniziale a una situazione in cui l'obiettivo è vero.

La dimostrazione di $\exists s\, onTable(A, s)$ data sopra è *costruttiva*: essa infatti termina con un'applicazione della regola di introduzione di $\exists$, dunque, esaminando la dimostrazione, si può estrarre un'istanza di situazione s per la quale vale $onTable(A, s)$, cioè $s = do(putOnTable(A), S_0)$. Possiamo interpretare questo termine come un piano che risolve il problema dato.

L'idea chiave della pianificazione deduttiva è che un piano soluzione può essere sintetizzato come *effetto collaterale della dimostrazione di teoremi.*

Così, in generale, pianificare nel calcolo delle situazioni, una volta assiomatizzato il dominio e lo stato iniziale, si riduce a dimostrare (costruttivamente) che esiste una situazione che soddisfa l'obiettivo $Goal(s)$:

$$Assiomi \vdash \exists s\, Goal(s)$$

Una prova costruttiva fornisce un'istanza S (cioè un termine di tipo situazione) tale che

$$Assiomi \vdash Goal(S)$$

Ad esempio, se nel mondo dei blocchi l'obiettivo è quello rappresentato in figura 2.3, la formula che lo rappresenta sarà

$$Goal(s) \equiv_{def} on(A, B, s) \wedge on(B, C, s) \wedge onTable(C, s)$$

Risolvere questo problema di pianificazione significa trovare una dimostrazione costruttiva di:

$$Assiomi \vdash \exists s\,(on(A, B, s) \wedge on(B, C, s) \wedge onTable(C, s))$$

Figura 2.3: Obiettivo nel mondo dei blocchi

Effettivamente, $\exists s\, Goal(s)$ è dimostrabile nella teoria del mondo dei blocchi e la soluzione estratta da una sua possibile dimostrazione è

$$S = do(putOn(A, B), do(putOn(B, C), do(putOnTable(A), S_0)))$$

Per estrarre la sequenza di azioni che costituisce il piano soluzione, le azioni vanno lette da destra verso sinistra: il piano soluzione è la sequenza: $putOnTable(A), putOn(B, C), putOn(A, B)$.

2.4 Esercizi

Per ciascuno dei seguenti domini, definire un linguaggio del calcolo delle situazioni adeguato per rappresentarlo e scrivere gli assiomi della corrispondente teoria: gli assiomi delle precondizioni, gli assiomi dello stato successore, gli assiomi del nome unico per gli oggetti e le azioni, l'assioma del dominio finito, eventuali assiomi per definire i predicati statici situazione iniziale, e, dove specificato, gli assiomi della situazione iniziale e la definizione della formula obiettivo.

1. Due robot, Pippo e Pluto. possono spostarsi tra diversi posti: cucina, salotto, camera da letto, giardino. L'unica azione possibile è "il robot r si sposta dal posto x al posto y.

2. Un robot si trova fuori di una stanza, la cui porta è chiusa. Nella stanza c'è un libro. Le azioni possibili sono (nella descrizione che segue omettiamo dai fluenti il parametro situazione):

   - *pass*: può essere eseguita solo se la porta è aperta (*open_door*), ed ha come effetto *in* (essere dentro la stanza), se l'agente si trova fuori, altrimenti (se è già dentro la stanza), l'effetto è $\neg in$.
   - *open_door*: possibile solo se la porta è chiusa, ha come effetto *open* (la porta è aperta);
   - *take_book*: possibile solo se il robot è dentro la stanza e non ha il libro in mano; ha come effetto *have_book*.

   (si noti che in questo problema i fluenti non hanno altri argomenti oltre al parametro situazione, e le azioni non hanno parametri).

3. Due robot, Pippo e Pluto, possono dipingere oggetti e spostarli da un posto a un altro. I colori a disposizione sono il rosso, il giallo e il verde, gli oggetti sono una palla, un cubo e una piramide. I posti sono la cantina e il giardino. Le azioni possibili sono: "il robot r dipinge l'oggetto x del colore y" e "il robot r sposta l'oggetto x dal posto y al posto z". Inizialmente sia i robot che gli oggetti sono in cantina, la palla è rossa, il cubo e la piramide sono verdi. L'obiettivo è quello di avere la palla e la piramide gialle, il cubo rosso, e tutti e tre gli oggetti devono stare in giardino.

4. Un robot può spostarsi da un posto a un altro, prendere oggetti e consumarli. Gli oggetti sono: latte, banane e gelato, e si trovano inizialmente tutti in frigorifero. L'obiettivo è quello di avere il frigorifero vuoto, ed il gelato in mano al robot.

5. Si hanno a disposizione 3 contenitori, inizialmente vuoti, $c1, c2, c3$, ed una brocca, di capacità superiore a quella di ciascun contenitore. L'obiettivo è quello di riempire i tre contenitori. Le azioni possibili sono: riempire la brocca dal rubinetto, svuotare la brocca nel lavandino, riempire un contenitore mediante travaso dalla brocca (possibile solo se la brocca è piena e il contenitore è vuoto).

6. Si hanno due scatole, $s1$ e $s2$, e una penna p. Nella situazione iniziale $s1$ e $s2$ sono entrambe chiuse, $s2$ è sopra $s1$ e p è sopra $s2$. L'obiettivo è quello di avere p dentro $s1$, $s2$ sopra $s1$ e $s1$ e $s2$ entrambe chiuse. Le azioni possibili sono:

   - mettere l'oggetto x dentro y (possibile se y è una scatola aperta e x è una penna),
   - aprire x (possibile se x è una scatola sulla quale non è collocato alcun oggetto),
   - chiudere x (possibile se x è una scatola sulla quale non è collocato alcun oggetto),
   - togliere x dalla superficie di y (possibile se x è sopra y e nessun oggetto è sopra x),
   - mettere l'oggetto x sopra y (possibile se y è una scatola e la superficie di x e quella di y sono entrambe libere).

7. L'agente è fuori della stanza, e deve entrare, oltrepassando due porte, p_1 e p_2, che sono chiuse. Le possibili posizioni dell'agente sono 3: a_1 (davanti alla porta p_1), a_2 (tra la porta p_1 e la porta p_2), a_3 (dentro la stanza). Alcune proprietà sono sempre vere: il fatto che p_1 collega a_1 e a_2 e il fatto che p_2 collega a_2 e a_3. Le azioni possibili sono $open(x, y, z)$ (aprire la porta x, che collega y con z, quando si è nella posizione y) e $pass(x, y, z)$ (passare oltre la porta x, dalla posizione y alla posizione z).

Capitolo 3

Il Linguaggio Golog

3.1 Introduzione

Il calcolo delle situazioni costituisce il fondamento logico di un linguaggio di programmazione di alto livello, il Golog, acronimo per *alGOl in LOGic* [2, 7]. Il Golog è stato sviluppato presso il *Cognitive Robotics Group* dell'Università di Toronto:

> The Cognitive Robotics group is concerned with endowing robotic or software agents with higher level cognitive functions that involve reasoning, for example, about goals, perception, actions, the mental states of other agents, collaborative task execution, etc. To do this, it is necessary to describe, in a language suitable for automated reasoning, enough of the properties of the robot, its abilities, and its environment, to permit it to make high-level decisions about how to act. The group has developed effective methods for representing and reasoning about the prerequisites and effects of actions, perception and other knowledge-producing actions, and natural events and actions by other agents. These methods have been incorporated into a logic programming language for agents called GOLOG (alGOl in LOGic). A prototype implementation of the language has been developed. Experiments have been conducted in using the language to build a high-level robot controller, some software agent applications (e.g. meeting scheduling), and more recently business process modeling tools. [1]

Il linguaggio Golog è pensato come un linguaggio adatto ad applicazioni quali sistemi di controllo di alto livello di robot o dispositivi meccanici, la programmazione di agenti software intelligenti, la modellazione e la simulazione di sistemi ad eventi discreti, ecc.

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.

Sul sito del gruppo [1], sono disponibili, oltre a numerose pubblicazioni, il codice di un interprete Golog scritto in SWI Prolog e Eclipse Prolog. Il Golog è stato esteso in modo da incorporare nel linguaggio la concorrenza ed eventi esogeni (ConGolog), nonché azioni di percezione (*sensing*) (IndiGolog). Tali estensioni del linguaggio permettono di progettare sistemi di controllo più flessibili, per agenti che operano in scenari complessi.

Il fondamento logico del Golog è costituito dal calcolo delle situazioni. Nel capitolo precedente si è visto come costruire una teoria logica per le azioni semplici, che costituisce un modello logico dell'ambiente in cui si trova ad operare il sistema software. In questo capitolo vedremo come è possibile definire nel calcolo delle situazioni anche *azioni complesse*, come la sequenza di azioni, le azioni condizionali, i cicli, le scelte non deterministiche e la definizione di procedure. La definizione di tali operazioni complesse nel calcolo delle situazioni costituisce appunto il linguaggio Golog.

Nei prossimi paragrafi vedremo come rappresentare in Prolog la situazione iniziale e come implementare la definizione delle azioni semplici e le sequenze di azioni semplici. Successivamente si vedrà come si definiscono le azioni complesse e come si implementano in Prolog, realizzando così un interprete Golog in Prolog. Faremo riferimento all"interprete del Golog in SWI Prolog (`golog_swi.pl`), che, come si è detto, si può trovare sul sito del Gruppo di Robotica Cognitiva di Toronto [1].

3.2 Rappresentazione della situazione iniziale

Per rappresentare la situazione iniziale si utilizzerà la costante Prolog `s0` – minuscolo. Il programma dovrà contenere clausole che definiscono ciò che è vero nella situazione iniziale.

Cconsideriamo il semplice caso del mondo dei blocchi in cui ci sono solo due blocchi, A e B, entrambi sul tavolo. La situazione iniziale si può rappresentare in Prolog mediante il seguente insieme di fatti (anche i nomi degli oggetti, che sono costanti Prolog, sono minuscoli):

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

All'insieme dei fluenti abbiamo aggiunto, per comodità, il fluente *clear*: $clear(x, s)$ significa che nessun blocco è sul blocco x nella situazione s.

Si noti che, per via dell'assunzione di mondo chiuso del Prolog, la definizione della situazione iniziale non include la specifica di ciè che è falso (quello che non è dimostrabile è falso), ma soltanto gli atomi positivi (si confrontino i fatti che definiscono la situazione inziale nel programma con la definizione della situazione iniziale data nel paragrafo 3.2).

Analogamente, in Prolog non è necessario specificare gli assiomi del nome unico: nomi e strutture sintatticamente diverse sono necessariamente diversi (non unficabili) in Prolog. Né è necessario rappresentare l'assioma del dominio

finito. Nei casi in cui sia necessario tipizzare gli oggetti, è sufficiente inserire l'insieme dei fatti (o regole) che definiscono il tipo degli oggetti stessi, ad esempio:

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

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

3.3 Le azioni primitive e le sequenze di azioni

Le azioni semplici, come quelle considerate negli esempi del capitolo precedente, costituiscono le "primitive" del Golog. Una delle caratteristiche fondamentali del linguaggio è proprio quella che le primitive sono definite dall'utente. La definizione delle primitive è costituita dagli assiomi delle precondizioni e dagli assiomi dello stato successore.

Consideriamo ad esempio il mondo dei blocchi visto nel capitolo precedente, con l'aggiunta del fluente *clear*. Le azioni semplici *putOn* e *putOnTable* sono definite dal seguente codice Prolog:

```
%  Action Precondition Axioms.
poss(putOn(X,Y),S) :- \+ on(X,Y,S), \+ X=Y, clear(X,S), clear(Y,S).
poss(putOnTable(X),S) :- \+ ontable(X,S), clear(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), not(A = putOn(Y,X)).
```

Gli assiomi delle precondizioni sono esattamente la riscrittura in Prolog degli assiomi visti nel paragrafo 2.2.2 – ricordando che una clausola Prolog `Head :- Tail` rappresenta la chiusura universale della formula $Tail \rightarrow Head$.

La rappresentazione Prolog degli assiomi dello stato successore non corrisponde invece agli assiomi del paragrafo 2.2.2. Le clausole rappresentano infatti formule della forma

$$\forall a \forall s \forall \overline{x}(G_R^+(a,s,\overline{x}) \vee (R(\overline{x},s) \wedge \neg G_R^-(a,s,\overline{x}))) \rightarrow R(\overline{x},do(a,s))$$

e non

$$\forall a \forall s \forall \overline{x}[Poss(a,s) \rightarrow (R(\overline{x},do(a,s)) \equiv G_R^+(a,s,\overline{x}) \vee (R(\overline{x},s) \wedge \neg G_R^-(a,s,\overline{x})))]$$

(dove $\overline{x}$ è una sequenza di variabili).

Le differenze sono dovute alle limitazioni del linguaggio clausale del Prolog. In primo luogo osserviamo che l'implementazione degli assiomi dello stato successore ignorano la precondizione $Poss(a,s)$. Di questo fatto si dovrà tenere in qualche modo conto nel seguito.

La seconda differenza consiste nel fatto che la doppia implicazione

$$R(\overline{x}, do(a,s)) \equiv G_R^+(a,s,\overline{x}) \vee (R(\overline{x},s) \wedge \neg G_R^-(a,s,\overline{x})))$$

è rimpiazzata dall'implicazione da destra a sinistra. In altri termini (ignorando la precondizione $Poss(a,s)$), la *definizione* di $R(\overline{x}, do(a,s))$:

$$(F) \quad \forall a \forall s \forall \overline{x} R(\overline{x}, do(a,s)) \equiv G_R^+(a,s,\overline{x}) \vee (R(\overline{x},s) \wedge \neg G_R^-(a,s,\overline{x}))$$

è rappresentata considerando soltanto la sua *parte "se"*.

In altri termini, la formula F costituisce proprio la semantica Prolog della definizione del predicato $R(\overline{x}, s')$, quando s' ha la forma $do(a,s)$ (si veda il paragrafo 1.4).

Dal punto di vista procedurale, quando nel programma è presente una sola clausola con la testa che unifica con `R(X1,...,Xn,do(A,S))`, il corpo di tale clausola costituisce l'unico modo di dimostrare un atomo della forma `R(X1,...,Xn,do(A,S))`, e quando un tale atomo non è dimostrabile, si assume che esso sia falso (assunzione del mondo chiuso).

Ignorando per il momento il problema della condizione $Poss(a,s)$ negli assiomi dello stato successore, proviamo ad utilizzare l'implementaizone Prolog delle azioni del mondo dei blocchi per risolvere un problema di pianificazione.

Come si è detto, nel calcolo delle situazioni, un problema di pianificazione si risolve dimostrando che

$$Assiomi \vdash \exists s\, Goal(s)$$

Assumiamo che, nel semplice esempio considerato nel paragrafo 2.2.1, l'obiettivo sia quello di avere il blocco A sopra il blocco B. Possiamo allora definire l'obiettivo mediante la clausola:

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

In linea teorica, eseguendo il programma costituito dalla definizione delle azioni, dello stato iniziale e dell'obiettivo, con il goal `goal(S)`, il Prolog dovrebbe fornire (come risultato della dimostrazione di `goal(S)` dal programma) un piano soluzione rappresentato dal legame per la variabile `S`. Ma ecco il risultato che si ottiene:

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

Qui si vedono chiaramente due problemi: il primo è che la sequenza di azioni "parte" da una generica situazione `_G241` anziché dalla situazione iniziale. In altri termini, si assume che qualunque variabile (`_G241`) possa corrispondere a una situazione: le situazioni cioè non sono solo i termini che si possono costruire a partire dalla situazione iniziale S_0 utilizzando la funzione *do*. Infatti non abbiamo rappresentato l'assioma di induzione.

Il secondo problema – che è dovuto al fatto che abbiamo ignorato la condizione di eseguibilità delle azioni – è che le azioni vengono eseguite in situazioni in cui non valgono le loro precondizioni. Ad esempio, dopo aver messo b su a non si può mettere a su b.

In termini generali, possiamo dire che abbiamo ottenuto come soluzione una sequenza di azioni che di fatto non è eseguibile.

Entrambi i problemi si risolvono considerando i piani come *azioni complesse* costituite da *sequenze di azioni primitive*. I piani possono essere cioè definiti induttivamente come segue:

- 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

Si definisce poi, nel calcolo delle situazioni, un nuovo simbolo di predicato Do, che ha come argomenti un piano e due situazioni: $Do(\pi, s, s')$ rappresenta il fatto che *eseguendo il piano π a partire dalla situazione s si raggiunge la situazione s'*. (Attenzione alla differenza tra maiuscole e minuscole: Do è un simbolo di predicato, mentre do è un simbolo di funzione).

Il predicato Do è definito induttivamente sui piani come segue:

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

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

Sequenze: se π_1 e π_2 sono piani, allora:

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

La definizione di Do costituisce la semantica nel calcolo delle situazioni delle azioni primitive e dell'azione complessa "sequenza di azioni primitive".

Si noti che un'espressione della forma $Do(\pi, s, s')$ non è una formula del calcolo delle situazioni, ma un'*abbreviazione* per una formula del calcolo delle situazioni. Ad esempio, nel mondo dei blocchi:

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

L'ultima espressione ottenuta è effettivamente una formula del calcolo delle situazioni per il mondo dei blocchi.

Come si vede, la definizione di Do tiene ora conto delle condizioni di eseguibilità delle azioni. Inoltre dà la possibilità di specificare non soltanto la situazione finale, ma anche quella di partenza. Un problema di pianificazione si può quindi risolvere dimostrando che:

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

(anche in assenza dell'assioma di induzione e ignorando le precondizioni delle azioni negli assiomi dello stato successore).

Passiamo ora a considerare come definire il predicato Do in Prolog. Definiamo innanzitutto un operatore per la sequenza di azioni:

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

In tal modo, ad esempio, la sequenza si azioni $[a : b : c]$ viene rappresentata dal termine Prolog `a:b:c`. Qui, come nel seguito, ci sono piccole differenze nelle notazioni utilizzate nella rappresentazione Prolog, rispetto al linguaggio simbolico astratto del calcolo delle situazioni. Una di queste è proprio quella del simbolo *Do*: dato che in Prolog non è possibile usare simboli di predicato che iniziano con una lettera maiuscola, nel codice si utilizza `do`, minuscolo, proprio come il simbolo funzionale. La differenza tra i due sarà chiara dal contesto.

Il codice Prolog per la definizione di tale predicato è il seguente:

```
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 noti che la prima clausola della definizione include la condizione "se a è un'azione primitiva": occorre dunque definire, per ciascun dominio, le azioni primitive. Nel caso del nostro esempio del mondo dei blocchi, il programma deve includere anche le clausole seguenti:

```
%  Primitive Action Declarations.
primitive_action(putOn(_,_)).
primitive_action(putOnTable(_)).
```

Possiamo dunque definire un predicato per la soluzione di problemi di pianificazione come segue:

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

e risolvere il semplice problema nel mondo dei blocchi:

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

Come ulteriore esempio, consideriamo il problema che si ottiene sostituendo le clausole che definiscono la situazione iniziale e l'obiettivo come segue:

```
/* 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)
```

Tuttavia l'approccio proposto fin qui non è sempre efficace, a causa della strategia di ricerca del Prolog (in profondità). Infatti, il motore di inferenza del Prolog potrebbe non trovare soluzioni, come nel caso seguente:

```
/* 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).
```

In altri termini, non ci si può affidare al motore di ricerca del Prolog per trovare piani soluzione, ma è necessario controllare esplicitamente, da programma, la ricerca del piano. Nel paragrafo 3.7 si vedrà come sia possibile farlo in Golog.

3.4 Le azioni complesse in Golog

Come si è detto, un programma Golog è un'azione complessa, che viene ridotta ad azioni primitive. Oltre alla sequenza di azioni, le azioni complesse includono i test, azioni non deterministiche, azioni condizionali, cicli e procedure.

La semantica delle azioni complesse è definita nel calcolo delle situazioni, esattamente come abbiamo definito la semantica delle azioni primitive e della sequenza di azioni nel paragrafo precedente.

Nella definizione di azioni complesse si usano simboli extralogici (come $while$, if, ecc.), che funzionano come *abbreviazioni* (o *macro* da espandere) per espressioni logiche nel linguaggio del Situation Calculus.

Le azioni complesse possono essere *non deterministiche*, nel senso che diverse esecuzioni della stessa azione possono risultare in situazioni differenti.

La definizione delle azioni complesse Golog consiste nella definizione induttiva del predicato Do, per il quale abbiamo già visto la clausola base e quella per le sequenze di azioni. Per definire il significato delle azioni complesse, si estende dunque la definizione del predicato Do a tutte le azioni: l'abbreviazione $Do(\delta, s, s')$ rappresenterà il fatto che *eseguendo l'azione complessa δ a partire dalla situazione s si può raggiungere la situazione s'*.

A causa del non determinismo, si può infatti avere $Do(s, \delta, s')$ e $Do(s, \delta, s'')$ con $s' \neq s''$.

La definizione induttiva di Do include i casi delle azioni primitive e delle sequenze di azioni (primitive o complesse), che ripetiamo qui di seguito, nella loro definizione simbolica e nell'implementazione Prolog:

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

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

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

Sequenze di azioni

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

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

Nel resto di questo paragrafo consideriamo gli altri casi della definizione induttiva, sia nella loro forma logica astratta che nell'implementazione Prolog.

3.4.1 Azioni di test

Un'azione di test consiste nel controllare se una data formula è vera nella "situazione corrente", cioè nella situazione ottenuta eseguendo le azioni del programma fino al punto in cui si esegue il test. Se la formula vale nella situazione corrente, allora l'esecuzione dell'azione ha successo e non modifica la situazione, altrimenti fallisce.

Le formule del calcolo delle situazioni contengono normalmente fluenti, con i loro parametri di tipo situazione. Un'azione di test controlla la verità di una formula nella situazione corrente, ma il programmatore non può sapere qual è la situazione corrente. Infatti il programma ne avrà una rappresentazione soltanto al momento dell'esecuzione. Per questo motivo, un'azione di test non si applica a una formula del calcolo delle situazioni, ma a una *pseudo-formula*: un'espressione ottenuta da una formula sopprimendo tutti gli argomenti di tipo situazione dai fluenti. La definizione delle azioni di test ripristina tali argomenti nel modo opportuno:

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

dove ϕ è una pseudo-formula e $\phi[s]$ è la formula corrispondente del calcolo delle situazioni, dove tutti i fluenti hanno s come parametro situazione.

Ad esempio:

$$\begin{array}{ll} \text{se} & \phi = \forall x\,(onTable(x) \wedge \neg on(x, A)) \\ \text{allora} & \phi[s] = \forall x\,(onTable(x, s) \wedge \neg on(x, A, s)) \end{array}$$

Come esempio di espansione di una macro della forma $Do(\phi?, s, s')$ consideriamo:

$$\begin{array}{l} Do(\exists x(onTable(x) \wedge clear(x))?, s, s') \equiv_{def} \\ \qquad\qquad \exists x(onTable(x, s) \wedge clear(x, s)) \wedge s = s' \end{array}$$

In Prolog le azioni di test sono rappresentate con la sintassi `?(F)`, dove `F` è una pseudo-formula costruita utilizzando gli operatori definiti come segue:

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

Per la negazione, si utilizza il segno "meno" e per le formule quantificate, termini della forma `all(Variabile,Formula)` e `some(Variabile,Formula)`. Attenzione: in una pseudo-formula di un programma Golog non devono occorrere variabili Prolog. Ad esempio, `all(x,onTable(x))` è una pseudo-formula, sulla quale si può eseguire un'azione di test. Ma non sarebbe corretto utilizzare un'espressione come `some(x,on(x,Y))`.

La clausola che definisce il predicato `do` per le azioni di test è la seguente:

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

che richiama il predicato `holds`, a sua volta definito come segue:

```
/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

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)).
```

Sulla definizione del predicato `holds` è opportuno fare qualche osservazione:

1. Il caso base (l'ultima clausola della procedura `holds`) si applica quando `A` non è una formula non atomica, cioè o è una formula atomica oppure è un goal Prolog.

   (a) Nel primo caso viene ripristinato l'argomento situazione del fluente, e questo richiede che il programmatore definisca il predicato `restoreSitArg`, per mezzo di una clausola per ogni fluente. Ad esempio, nel dominio dei blocchi si dovranno avere le clausole:

   ```
   restoreSitArg(onTable(X),S,onTable(X,S)).
   restoreSitArg(on(X,Y),S,on(X,Y,S)).
   restoreSitArg(clear(X),S,clear(X,S)).
   ```

   Tentando di dimostrare `holds(A,F)`, dunque, si tenta innanzitutto di ripristinare la situazione `S` nella pseudo-formula `A`; se questo ha successo (perché `A` è un fluente e il programma contiene l'apposita

```
%  Consult the Golog interpreter
:- [golog_swi].

%  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).

%  Primitive Action Declarations.
primitive_action(putOn(_,_)).
primitive_action(putOnTable(_)).

%  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)).

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

Figura 3.1: Un semplice esempio nel dominio dei blocchi

clausola `restoreSitArg`) e si ottiene la formula (atomica) `F`, questa viene invocata come goal Prolog.

Ad esempio, si consideri l'implementazione del mondo dei blocchi il cui codice è riportato per intero in figura 3.1, con situazione iniziale:

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

Se si invoca il goal `holds(onTable(b),do(putOnTable(b),s0))`, viene applicata la clausola base della definizione di `holds`, ed il Prolog tenta quindi innanzitutto di soddisfare:

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

Il primo atomo è dimostrabile, con il 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.

(b) Se il caso (a) non ha successo, allora si controlla che `A` non sia uno pseudo-fluente (`restoreSitArg(A,S,F)` fallisce) né una pseudo-formula complessa (`isAtom(A)`). In caso di successo, viene invocato `A` come goal Prolog. Ciò significa, di fatto, che il Golog estende il Prolog, nel senso che può utilizzare ogni predicato Prolog (indipendente dalla situazione `S`).

Ad esempio:

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

2. Consideriamo il caso del quantificatore esistenziale:

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

dove il predicato `sub` è definito come segue:

```
/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name
   replaced by New. */
sub(_,_,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- \+ var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :-
     \+ T1 = X1, T1 =..[F|L1],
     sub_list(X1,X2,L1,L2),
     T2 =..[F|L2].
sub_list(_,_,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :-
     sub(X1,X2,T1,T2),
     sub_list(X1,X2,L1,L2).
```

Quindi `sub(V,_,P,P1)` istanzia `P1` all'espressione che si ottiene da `P` sostituendo ogni occorrenza di `V` con una variabile Prolog nuova.

Ad esempio, quando si dimostra il goal `holds(some(x,onTable(x)),s0)` nel semplice esempio del mondo dei blocchi considerato sopra, viene invocato il goal `sub(x,_,onTable(x),P1)`, che ha successo con (ad esempio) `P1 = onTable(_G176)`. Viene allora invocato, ricorsivamente:
`holds(onTable(_G176),s0)`;
questo si riduce al goal Prolog `onTable(_G176,s0)`, che ha successo.

3. 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 del mondo rappresentato nella figura 3.1, esiste un blocco che non è sul tavolo, ma:

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

Infatti, `holds(some(x,-onTable(x)),s0)` si riduce a `holds(-onTable(X),s0)` (dove `X` è una qualsiasi variabile Prolog nuova). Ci troviamo dunque in un caso in cui `holds` è invocato sulla negazione di una pseudo-formula contenente una variabile Prolog non legata. Per il caso della negazione, tale goal si riduce a `\+ holds(onTable(X),s0)`. Ma `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)`.

4. Consideriamo ora i casi del quantificatore universale:

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

Nella situazione iniziale del mondo dei blocchi della figura 3.1, il blocco `b` non è sul tavolo, quindi non tutti i blocchi sono sul tavolo. Eppure:

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

Per capirne il motivo, occorre considerare l'osservazione precedente sull'uso della negazione come fallimento. Il goal `holds(-all(x,onTable(x)),s0)` si riduce a `holds(some(x,-onTable(x)),s0)`, che, come abbiamo visto nel punto precedente, fallisce.

Per lo stesso motivo `holds(all(x,onTable(x)),s0)` ha successo, dato che `holds(some(x,-onTable(x)),s0)` fallisce.

Occorre dunque prestare molta attenzione anche all'uso di test contenenti il quantificatore universale.

I problemi dovuti all'uso della negazione come fallimento si possono evitare aggiungendo al programma la definizione del tipo degli oggetti sui quali si vuole quantificare, nel nostro caso:

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

ed utilizzando soltanto formule con quantificazione limitata, cioè della forma $\forall x\,(T(x) \rightarrow A(x))$ e $\exists x\,(T(x) \wedge A(x))$, dove $T(x)$ è la formula atomica che rappresenta il tipo di x. Nel nostro esempio, il comportamento (corretto) che otteniamo è il seguente:

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

Consideriamo infatti il primo caso: `holds(some(x,block(x) & -onTable(x)),s0)` si riduce a `holds(block(X) & -onTable(X),s0)`, che a sua volta si riduce a

`holds(block(X),s0), holds(-onTable(X),s0)`. Il successo del primo dei due atomi della congiunzione vincola `X` ad `a` oppure a `b`, quindi quando viene invocato `holds(-onTable(X),s0)`, la variabile `X` è legata (a una costante) e la negazione come fallimento si comporta correttamente.

Analogamente, `holds(all(x,block(x) => onTable(x)),s0)` si riduce a `holds(-some(x,-(block(x) => onTable(x))),s0)`, che ha successo se e solo se `holds(some(x,-(block(x) => onTable(x))),s0)` fallisce. Tale goal, a sua volta, si riduce a `holds(block(X) & -onTable(X),s0)`, che, come abbiamo visto, si comporta correttamente perché quando si tenta di soddisfare `holds(-onTable(X),s0)`, la variabile `X` è legata a una costante.

E anche nel terzo caso, quando viene invocato `holds` su un atomo negato, le variabili che esso contiene sono tutte completamente istanziate.

3.4.2 Scelte non deterministiche

Una delle caratteristiche che rendono il Golog un linguaggio di livello più alto rispetto ad altri linguaggi di programmazione è il non determinismo.

Scelta non deterministica fra due azioni

Il caso più semplice di non determinismo consiste nella scelta non deterministica tra due azioni: se δ_1 e δ_2 sono azioni (semplici o complesse), $(\delta_1|\delta_2)$ è l'azione che consiste nell'esecuzione di δ_1 *oppure* (non deterministicamente) δ_2. Nella definizione ricorsiva di Do la clausola che definisce la scelta tra due azioni è la seguente:

$$Do((\delta_1 \mid \delta_2), s, s') \equiv_{def} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

Dunque $Do((\delta_1 \mid \delta_2), s, s')$ è vero se vale una delle due formule $Do(\delta_1, s, s')$ o $Do(\delta_2, s, s')$

Come esempio di espansione di una macro della forma $Do((\delta_1 \mid \delta_2), s, s')$, consideriamo, nel mondo dei blocchi:

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

In Prolog la scelta non deterministica tra due azioni è rappresentata mediante la sintassi `E1 # E2`. L'operatore `#` è definito dalla direttiva:

```
:- op(960, xfy, [#]).   /* Nondeterministic action choice */
```

e la scelta non deterministica mediante la clausola:

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

Ad esempio, nel mondo della figura 3.1, 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.
```

Ovviamente, se la prima azione non è eseguibile (`do(E1,S,S1)` fallisce), la scelta deterministicamente cade sulla seconda; e viceversa. Ad esempio, sempre nel mondo della figura 3.1:

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

Scelta non deterministica degli argomenti di un'azione

Un'altra forma di non determinismo consiste nella scelta dell'argomento di un'azione: $(\pi x)\delta(x)$ è l'azione che consiste nello scegliere un argomento x ed eseguire con esso l'azione $\delta(x)$. La scelta non deterministica degli argomenti di un'azione è definita come segue:

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

L'azione è dunque eseguibile se $\delta(x)$ è eseguibile per qualche x.

Come esempio di espansione di una macro della forma $Do((\pi x)\delta(x)), s, s')$, consideriamo, nel mondo dei blocchi:

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

Quindi l'azione $(\pi x)\, putOnTable(x)$ è quella di scegliere un blocco x per il quale sia possibile eseguire $putOnTable(x)$, ed eseguire tale azione. Nel caso sia possibile eseguire $putOnTable(x)$ per diversi x, si avranno diverse scelte possibili, quindi diversi modi di eseguire l'azione.

Come ulteriore esempio, consideriamo un mondo in cui un robot si può muovere da x a y mediante l'azione *goto(x,y)*. L'azione complessa $\gamma = (\pi x)\, goto(c, x)$ (dove c è una costante) rappresenta l'azione del robot di muoversi in un luogo qualunque x ($x \neq c$, se richiesto dalle precondizioni di *goto*). L'espansione della macro $Do((\pi x)\, goto(c, x), s, s')$ è la seguente:

$$\begin{array}{rll} Do((\pi x)\, goto(c, x), s, s') & \equiv_{def} & \exists x\, Do(goto(c, x), s, s') \\ & \equiv_{def} & \exists x\, (Poss(goto(c, x), s) \wedge s' = do(goto(c, x), s)) \end{array}$$

In Prolog la scelta non deterministica di un argomento di un'azione è rappresentata mediante la sintassi `pi(V,E)` ed è definita come segue:

```
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
```

Ricordiamo che `sub(Name,New,Term1,Term2)` ha successo se `Term2` si ottiene da `Term1` sostituendo `Name` con `New`.

Come nel caso delle variabili nelle formule, occorre far attenzione a non confondere le variabili Golog (minuscole) con le variabili Prolog. Ad esempio, `pi(x,putOnTable(x))` è un'azione Golog, e non sarebbe corretto scrivere `pi(X,putOnTable(X))`.

Come esempio, consideriamo sempre il programma della figura 3.1, con l'aggiunta delle clausole

```
onTable(c,s0).
clear(c,s0).
onTable(d,s0).
clear(d,s0).
```

Si hanno allora due scelte possibili per l'esecuzione dell'azione `pi(x,putOn(x,c)` nella situazione iniziale:

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

E sei possibili scelte per coppie di argomenti di `putOn`:

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

Si noti che il goal `do(pi(x,putOn(x,c),s0,S)` si riduce, per definizione, al goal `do(putOn(X,c),s0,S)`, dove `X` è una qualsiasi variabile nuova. Come al solito, occorre fare molta attenzione all'interazione con la negazione come fallimento del Prolog, dato che l'azione `putOn(X,c)` contiene una variabile Prolog. In questo caso si tratta di un'azione atomica, dunque il goal `do(putOn(X,c),s0,S)` si riduce alla congiunzione dei due goal `primitive_action(putOn(X,c))` (che ha successo), e `poss(putOn(X,c),s0)`. Ricordiamo la clausola per `putOn` della definizione di `poss`:

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

Il corpo della clausola contiene atomi negati, ed è importante che, ogniqualvolta questi vengono richiamati, tutte le variabili siano istanziate. Nel nostro caso, tentando di dimostrare il goal `poss(putOn(X,c),S)`, la variabile `X` viene istanziata, mediante la chiamata a `clear(X,S)`, prima di tentare di dimostrare le negazioni `\+ on(X,c,S)` e `\+ X=Y`.

Se avessimo invece definito `poss` invertendo l'ordine degli atomi nel corpo della clausola:

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

la scelta non deterministica degli argomenti dell'azione non funzionerebbe come ci si aspetta:

```
?- do(pi(x,putOn(x,c)),s0,S).
false.
```

Infatti in questo caso, quando si "esegue" il goal `\+ X=c`, `X` non è istanziata: quindi `X=c` ha successo e `\+ X=c` fallisce.

3.4.3 Iterazione

Se δ è un'azione (semplice o complessa), l'azione δ^* consiste nell'esecuzione di δ zero o più volte. La scelta del numero di iterazioni è non deterministica.

Di conseguenza, la relazione $Do(\delta^*, s, s')$ è la chiusura riflessiva e transitiva della relazione $Do(\delta, s, s')$. In altri termini, $Do(\delta^*, s, s')$ si può definire induttivamente come segue:

- se $s = s'$ allora $Do(\delta^*, s, s')$ (chiusura riflessiva);
- se $Do(\delta, s, s')$ e $Do(\delta^*, s', s'')$, allora $Do(\delta^*, s, s'')$ (chiusura transitiva).

La definizione dell'iterazione in Prolog è semplice. Essa è rappresentata dal costrutto `delta(E)` e la corrispondente clausola nella definizione di `do` è la seguente:

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

Questa clausola definisce correttamente in Prolog tutte le *istanze positive* della relazione `do(star(E),S,S1)`. Si ricordi infatti quel che si è detto alla fine del paragrafo 1.4.

Per quel che riguarda la definizione dell'iterazione nel calcolo delle situazioni, dobbiamo ricordare che la chiusura transitiva di un relazione non è definibile in logica del primo ordine. Quindi, la definizione della chiusura riflessiva e transitiva di $Do(\delta, s, s')$ richiede la logica del secondo ordine (vedi paragrafo 1.3.3):

$$\begin{aligned} Do(\delta^*, s, s') \equiv_{def} & \\ \forall P\{ \quad & \forall s_1 P(s_1, s_1) \wedge \\ & \forall s_1, s_2, s_3 \, [Do(\delta, s_1, s_2) \wedge \; P(s_2, s_3) \rightarrow P(s_1, s_3) \,] \\ & \quad \rightarrow P(s, s') \,\} \end{aligned}$$

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 l'esecuzione di δ in s_1 porta in s_2 e $(s_2, s_3) \in P$, 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.

3.4.4 Istruzioni condizionali e cicli

Istruzioni condizionali e cicli *while* possono essere definiti usando i costrutti precedentemente introdotti. L'istruzione condizionale (`if/then/else`) è definita come segue:

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

Essa equivale all'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.

La definizione della corrispondente clausola del predicato `do` in Prolog è la seguente:

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

I cicli `while` sono definiti come segue:

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

Un ciclo **while** ϕ **do** δ **endWhile** equivale alla sequenza di due istruzioni:

- la prima, $[\phi?; \delta]^*$, consiste nell'iterazione (zero o più volte) della sequenza $[\phi?; \delta]$ (azione test ϕ seguita dall'azione δ); se ϕ è falsa nella situazione corrente, l'azione di test fallisce, quindi δ non viene eseguita;

- la seconda, $\neg\phi?$, è l'azione di test su $\neg\phi$, che ha successo quando ϕ è falsa nella situazione corrente; il test garantisce che l'uscita dal ciclo si abbia soltanto quando ϕ è falsa.

In Prolog, le istruzioni di ciclo sono rappresentate da termini della forma `while(P,E)`, dove `P` è la condizione e `E` l'azione, e sono definite come segue:

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

Come esempio di utilizzazione del costrutto `while`, si consideri il dominio del mondo dei blocchi rappresentato in Prolog come nella figura 3.1, ma con la seguente situazione iniziale:

```
%  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).
```

e con l'aggiunta della definizione del predicato `block`:

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

Il seguente goal Prolog invoca l'esecuzione dell'azione "sposta un blocco sul tavolo finché possibile":

```
?- 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)))
```

3.4.5 Procedure

Un programma Golog può essere strutturato in un insieme di *procedure*: il programma può contenere una sequenza di *dichiarazioni di procedure*, che possono anche essere definite ricorsivamente e che possono richiamare altre procedure.

La dichiarazione di una procedura P specifica i parametri formali $v_1, ..., v_n$ e il corpo della procedura δ, nella forma seguente:

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

Una chiamata di procedura in un programma ha la forma $P(t_1, ..., t_n)$, e $do(P(t_1, ..., t_n), s, s')$ significa che eseguendo la procedura P con parametri attuali $t_1, ..., t_n$ si ottiene la transizione dalla situazione s alla situazione s'.

La dichiarazione di una procedura in Prolog è un fatto della forma:

```
proc(P(V1,...,Vn),Body(V1,...,Vn)).
```

Ad esempio possiamo definire nel mondo dei blocchi una procedura (senza parametri) che mette tutti i blocchi sul tavolo (assumendo che sia definito il predicato `block`, come ad esempio a pagina 47):

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

Se la situazione iniziale è quella con 5 blocchi, descritta a pagina 47, l'esecuzione della procedura nella situazione iniziale produrrà la situazione seguente:

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

Una procedura con parametro X che smonta la torre che ha il blocco X in cima può essere definita ricorsivamente come segue:

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

Smontare la torre che ha il blocco c in cima nella situazione di pagina 3.4.4 produce la situazione seguente:

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

Osservazione importante: i parametri di una procedura sono variabili Prolog (iniziano con lettera maiuscola), mentre le variabili di un operatore π e quelle dei quantificatori sono variabili Golog, e non variabili Prolog (iniziano con lettera minuscola).

In Prolog la semantica della chiamata di procedure è definita dalla seguente clausola per il predicato `do`:

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

Nel Situation Calculus anche la semantica di procedure ricorsive 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 $Do(E(t_1, ..., t_n), s, s')$.

3.5 Esempi

3.5.1 Controllo di un ascensore

Questo esempio è tratto da [7]. Il problema consiste nella modellizzazione e gestione di un ascensore, che opera in un edificio con un determinato numero di piani. Le azioni semplici sono: salire o scendere di un piano, aprire e chiudere le porte (tutte senza parametri).

I fluenti utilizzati sono: `currentFloor(N,S)` (N è il piano a cui si trova l'ascensore nella situazione S) e `on(N,S)` (nella situazione S è acceso il bottone di chiamata al piano N).

Il codice Prolog seguente rappresenta la conoscenza sui predicati statici, gli assiomi delle precondizioni, gli assiomi dello stato successore e la descrizione della situazione iniziale.

```
        %  THE SIMPLIFIED ELEVATOR CONTROLLER

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

% Primitive control actions
primitive_action(up).   % PutOn elevator 1 floor up
primitive_action(down). % PutOn 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,_).

% 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)).
```

In questo contesto, è possibile definire, ad esempio, due procedure, `down(N)` e `up(N)`, che, quando eseguite, fanno scendere o salire l'ascensore di N piani:

```
% 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))).
```

Val la pena osservare, in questi esempi, l'interazione tra Golog e Prolog: un predicato Prolog (predefinito o definito dall'utente) può essere utilizzato nelle azioni di test. Infatti il "caso base" della definizione del predicato `holds` determina che `holds(A,S)` vale quando `A`, che non è un una pseudo-formula, è dimostrabile come goal Prolog (si veda pagina 41).

Ecco alcuni esempi di esecuzione delle proceure:

```
?- 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
```

Come ulteriori esempi, possiamo definire una procedura `goFloor(N)` che, quando eseguita, porta l'ascensore al piano N:

```
% 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))))
```

Infine, se l'obiettivo del "programma principale" di controllo dell'ascensore è quello di servire tutte le chiamate dell'ascensore (cioè portare l'ascensore a un piano dove è stato chiamato, e lì aprire e chiudere la porta), definiamo due procedure ausiliarie: `serve(N)` che serve il piano N, e `serveAfloor` che serve un qualsiasi piano dove sia stato chiamato l'ascensore, ed infine la procedura principale:

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

?- 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)))

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

3.5.2 Controllo di un robot con un unico braccio

Consideriamo un semplice dominio in cui un singolo robot, con un solo braccio, può prendere e lasciare oggetti, spostandoli da un tavolo al pavimento, o viceversa. Le azioni semplici sono prendere un oggetto, e lasciare sul tavolo o sul pavimento un oggetto che il robot ha in mano. Nella situazione iniziale il robot non tiene nulla in mano e ci sono due oggetti sul tavolo, a e b.

Il dominio può essere rappresentato come segue:

```
% direttive iniziali
:- [golog_swi].
:- discontiguous onTable/2, armempty/1.

% 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
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).

% Assiomi dello 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)).
```

La seguente procedura `clearTable` sgombra il tavolo da tutti gli oggetti che vi si trovano (assumendo che, quando richiamata, il robot non abbia nulla in mano). Essa richiama la procedura ausiliaria `removeFromTable(X)`, che, quando eseguita, toglie l'oggetto `X` dal tavolo e lo pone per terra.

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

3.5.3 Controllo di robot in movimento

Si consideri un dominio in cui alcuni robot possono muoversi in un ambiente costituito da un certo numero di stanze, tutte collegate a un corridoio; alcune stanze sono collegate tra loro anche direttamente. I robot possono inoltre prendere oggetti e depositare oggetti. Le azioni possibili sono: *move(r,x,y)* (il robot r va dal luogo x al luogo y – possibile solo se x e y sono collegati), *take(r,x,y)* (il robot r prende l'oggetto x, trovandosi nel luogo y – i robot possono tenere in mano un qualsiasi numero di oggetti) e *drop(r,x,y)* (il robot r lascia l'oggetto x nel luogo y). Oltre a predicati statici (per la tipizzazione degli oggetti e la rappresentazione della topologia dell'ambiente), la descrizione del dominio si avvale dei seguenti fluenti: *atR* (per rappresentare la posizione dei robot) e *at*

(che rappresenta la posizione degli oggetti, in un luogo o in mano a un robot). Nella situazione iniziale ci sono due robot, Pippo e Pluto, 6 stanze (1,...,6) e 6 oggetti (libro, quaderno, penna, matita, cellulare, lampada). La topologia delle stanze è rappresentata in figura 3.2: ogni stanza è collegata direttamente a quelle adiacenti e al corridoio. I due robot sono inizialmente nella stanza 1 e gli oggetti tutti nella stanza 4.

1	3	5
corridoio		
2	4	6

Figura 3.2: La topologia delle stanze

Il dominio può essere rappresentato in Golog come segue:

```
% direttive iniziali
:- [golog_swi].
:- discontiguous atR/3, at/3.

% dichiarazione delle azioni primitive
primitive_action(move(_,_,_)).
primitive_action(take(_,_,_)).
primitive_action(drop(_,_,_)).

% Restore suppressed situation arguments.
restoreSitArg(atR(R,P),S,atR(R,P,S)).
restoreSitArg(at(X,Y),S,at(X,Y,S)).

% definizione di eventuali predicati statici
connesso(X,corridoio):- stanza(X).
connesso(corridoio,X) :- stanza(X).
connesso(X,Y) :- stanza(X), stanza(Y), (X is Y+2 ; X is Y-2).
robot(pippo). robot(pluto).
stanza(X) :- member(X,[1,2,3,4,5,6]).
oggetto(X) :- member(X,[libro,quaderno,penna,matita,cellulare,lampada]).

% Action precondition axioms
poss(move(R,X,Y),S) :- atR(R,X,S), connesso(X,Y).
poss(take(R,X,P),S) :- atR(R,P,S), at(X,P,S).
poss(drop(R,X,P),S) :- atR(R,P,S), at(X,R,S).

% Successor state axioms
atR(R,P,do(A,S)) :-
        A=move(R,_,P) ;
        atR(R,P,S), \+ A=move(R,P,_).
at(X,Y,do(A,S)) :-
        A=take(Y,X,_); A=drop(_,X,Y);
        at(X,Y,S), \+ (A=take(_,X,Y); A=drop(Y,X,_)).
```

```
% Initial situation
atR(X,1,s0) :- robot(X).
at(X,4,s0) :- oggetto(X).
```

Si vuole scrivere una procedura `porta(R,X,P)`, che, quando eseguita (assumendo che il robot R non abbia già in mano l'oggetto X), fa sì che il robot R vada a prendere l'oggetto X e lo porti nella stanza P. La soluzione qui proposta utilizza una procedura ausiliaria, `va(R,P)`, che sposta il robot R dal luogo in cui si trova alla stanza P.

```
%% procedura ausiliaria che sposta il robot R nel posto P
proc(va(R,P),
     ?(atR(R,P)) #
    (?(-atR(R,P)):
    pi(p,?(atR(R,p)):
       (move(R,p,P) # (move(R,p,corridoio):move(R,corridoio,P))))).

%%% si assume che R non abbia X in mano
proc(porta(R,X,P),
     at(X,P) #
     (?(-at(X,P)):
     pi(x, ?(at(X,x)):va(R,x):
       take(R,X,x) :
       va(R,P) : drop(R,X,P)))).

?- do(porta(pippo,libro,6),s0,S).
S = do(drop(pippo, libro, 6), do(move(pippo, 4, 6),
       do(take(pippo, libro, 4), do(move(pippo, corridoio, 4),
       do(move(pippo, 1, corridoio), s0)))))
```

3.5.4 L'ora del tè

Questo esempio è una semplificazione del *teatime domain*, un problema classico in pianificazione in intelligenza artificiale.

Un robot può muoversi in un ambiente costituito da un certo numero di stanze, tutte collegate a un corridoio centrale. Stanze adiacenti hanno anche una porta che le collega direttamente. In una delle stanze è collocata una pila di tazze e in un'altra c'è la macchina per fare il tè. Alcune delle stanze possono richiedere il tè e il robot deve allora servirle.

Per rappresentare il dominio, utilizziamo i seguenti fluenti: `atRobby(X,S)` (il robot si trova nella locazione `X`), `freeRobby(S)` (il robot ha le mani libere), `emptycuploaded(S)` (il robot ha in mano una tazza vuota), `fullcuploaded(S)` (il robot ha in mano una tazza piena), e `ordered(X,S)` (la stanza `X` ha ordinato il tè).

Le azioni possibili per il robot sono: `getcup(X)` (prendere una tazza nella stanza `X`), `fillcup(X)` (riempire, nella stanza `X`, la tazza vuota che il robot ha in mano), `deliver(X)` (consegnare la tazza di tè nella stanza `X`) e `goto(X,Y)` (andare dalla locazione `X` alla locazione `Y`).

Nel problema considerato ci sono 4 stanze; le stanze 1 e 3 sono adiacenti, ed anche le stanze 2 e 4; le tazze si trovano nella stanza 1, la macchina del tè nella stanza 2; tutte le stanze hanno inizialmente ordinato il tè.

Possiamo rappresentare il dominio come segue:

```
% direttive iniziali
:- [golog_swi].
:- discontiguous atRobby/2, freeRobby/1, ordered/2.

% static predicates
connected(room(1),hallway).
connected(room(2),hallway).
connected(room(3),hallway).
connected(room(4),hallway).
connected(room(1),room(3)).
connected(room(2),room(4)).
teamachine(room(1)).
cupstack(room(2)).

%  Action Precondition Axioms.
poss(go(X,Y),S) :- (connected(X,Y) ; connected(Y,X)),
                   atRobby(X,S).
poss(getcup(X),S) :- atRobby(X,S), cupstack(X), freeRobby(S).
poss(fillcup(X),S) :- atRobby(X,S), teamachine(X), emptycuploaded(S).
poss(deliver(X),S) :- atRobby(X,S), ordered(X,S), fullcuploaded(S).

%  Successor State Axioms.
atRobby(X,do(A,S)) :- A = go(_,X) ; (atRobby(X,S), \+ A = go(X,_)).
freeRobby(do(A,S)) :- A = deliver(_) ; (freeRobby(S), \+ A=getcup(_)).
emptycuploaded(do(A,S)) :-
        A = getcup(_) ; (emptycuploaded(S), \+ A=fillcup(_)).
fullcuploaded(do(A,S)) :-
        A = fillcup(_) ; (fullcuploaded(S), \+ A=deliver(_)).
ordered(X,do(A,S)) :- ordered(X,S), \+ A=deliver(X).

%  Primitive Action Declarations.
primitive_action(getcup(_)).
primitive_action(fillcup(_)).
primitive_action(deliver(_)).
primitive_action(go(_,_)).

% Restore suppressed situation arguments
restoreSitArg(atRobby(X),S,atRobby(X,S)).
restoreSitArg(freeRobby,S,freeRobby(S)).
restoreSitArg(emptycuploaded,S,emptycuploaded(S)).
restoreSitArg(fullcuploaded,S,fullcuploaded(S)).
restoreSitArg(ordered(X),S,ordered(X,S)).

%  Initial State
atRobby(room(1),s0).
ordered(room(1),s0).
ordered(room(2),s0).
ordered(room(3),s0).
```

```
ordered(room(4),s0).
freeRobby(s0).
```

La procedura Golog `serve_all_rooms`, definita qui di seguito, guida il robot a consegnare il tè in tutte le stanze in cui è stato ordinato. Essa utilizza la procedura `serve(X)` per servire il tè nella stanza `X`. Questa, a sua volta, richiama la procedura `goto(X)`, che sposta il robot dal luogo in cui si trova in `X`.

```
% Procedure
proc(serve_all_rooms,
     while(some(r,ordered(r)),
           pi(r, ?(ordered(r)):serve(r)))).
proc(serve(X),
     pi(r,?(cupstack(r)):
       goto(r):getcup(r)):
    pi(r,?(teamachine(r)):
      goto(r):fillcup(r)):
    goto(X):deliver(X)).
proc(goto(X),
     ?(atRobby(X)) #
     pi(from,?(atRobby(from)):
       (go(from,X) # go(from,hallway):go(hallway,X)))).
```

Ecco un esempio di esecuzione della procedura `goto`:

```
?- do(goto(room(3)),s0,S).
S = do(go(room(1), room(3)), s0) ;
S = do(go(hallway, room(3)), do(go(room(1), hallway), s0))
```

L'esecuzione delle altre procedure richiede in generale un discreto numero di azioni primitive. Può essere dunque utile definire un predicato Prolog per "compilare" un programma Golog, in modo da ottenere una stampa della sequenza di azioni primitive che ne costituiscono un'esecuzione.

```
% compilazione di un programma Golog
compile(P) :-
        do(P,s0,S),
        makeActionList(S,Alist),
        nl, writelist(Alist).

% Action sequences
makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1),
                              append(L1, [A], L).

% writelist
writelist([]) :- nl.
writelist([X|Rest]) :- write(X), nl, writelist(Rest).
```

Ecco allora alcuni esempi di esecuzione delle procedure sopra definite:

```
?- compile(serve(room(3))).

go(room(1), hallway)
go(hallway, room(2))
getcup(room(2))
go(room(2), hallway)
go(hallway, room(1))
fillcup(room(1))
go(room(1), room(3))
deliver(room(3))

true

?- compile(serve_all_rooms).

go(room(1), hallway)
go(hallway, room(2))
getcup(room(2))
go(room(2), hallway)
go(hallway, room(1))
fillcup(room(1))
deliver(room(1))
go(room(1), hallway)
go(hallway, room(2))
getcup(room(2))
go(room(2), hallway)
go(hallway, room(1))
fillcup(room(1))
go(room(1), hallway)
go(hallway, room(2))
deliver(room(2))
getcup(room(2))
go(room(2), hallway)
go(hallway, room(1))
fillcup(room(1))
go(room(1), room(3))
deliver(room(3))
go(room(3), hallway)
go(hallway, room(2))
getcup(room(2))
go(room(2), hallway)
go(hallway, room(1))
fillcup(room(1))
go(room(1), hallway)
go(hallway, room(4))
deliver(room(4))

true
```

3.5.5 Schema generale

Per riassumere, può essere utile considerare tutti gli elementi che devono essere presenti in un file che formalizza un determinato dominio nel calcolo delle situazioni, in modo che possa essere utilizzato dal Golog. Lo schema generale è il seguente (ogni "sezione" è accompagnata da un esempio, commentato):

```
%%%%%%%%%%%%%%% direttive iniziali
%% caricare in memoria l'interprete Golog
:- [golog_swi].

%%%%%%%%%%%%%%% direttive discontiguous per i fluenti
%%% esempio
%%% :- discontiguous currentFloor/2, on/2.

%%%%%%%%%%%%%% dichiarazione delle azioni primitive
%% esempio
%% primitive_action(move(_,_)).

%%%%%%%%%%%%%% Restore suppressed situation arguments.
%% (un fatto per ogni fluente).
%% esempio
%% restoreSitArg(on(X,Y),S,on(X,Y,S)).

%%%%%%%%%%%%%% definizione di eventuali predicati statici
%% esempio
%% top(10).

%%%%%%%%%%%%%% Action precondition axioms
%% esempio
%% poss(move(X,Y),S) :- clear(X,S), clear(Y,S), not(X = Y).

%%%%%%%%%%%%%% Successor state axioms
%% esempio
%% on(X,Y,do(A,S)) :-
%%        A = move(X,Y) ;
%%        on(X,Y,S), not(A = moveToTable(X)), not(A = move(X,_)).

%%%%%%%%%%%%%% Initial situation
%% esempio
%% on(b,c,s0). on(c,a,s0). on(a,d,s0). ontable(d,s0). clear(b,s0).
```

3.6 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. Nel dominio di controllo dell'ascensore, definire una procedura $park$ che posiziona l'ascensore a piano terra.
4. 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.
5. 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.

3.7 Pianificazione in Golog

In questa sezione descriveremo come affrontare il problema della pianificazione atomatica utilizzando il linguaggio Golog.

In linea teorica, un problema di pianificazione si potrebbe semplicemente risolvere cercando una qualsiasi sequenza di azioni (o, in generale, un'azione complessa Golog) che conduca dalla situazione iniziale a una situazione obiettivo. Ma, impostando il problema in questo modo semplice, il motore di inferenza del Prolog può non trovare soluzioni.

Si assuma di restringere la definizione delle azioni complesse alla sola sequenza di azioni (un piano è infatti una sequenza di azioni), cioè di definire il predicato `do` mediante le due seguenti clausole soltanto:

```
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).
```

E si consideri il mondo dei blocchi con la seguente situazione iniziale e obiettivo:

```
/* 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).
```

In linea teorica, dovrebbe essere possibile risolvere il problema di pianificazione dimostrando che esiste una sequenza di azioni `P` che, quando eseguita a partire dalla situazione iniziale, porta a una situazione che soddisfa il predicato `goal`:

```
?- do(P,s0,S), goal(S).
```

Ma l'esecuzione del programma su questo goal non termina.

Ciò è dovuto al meccanismo di ricerca del Prolog, che opera in profondità. Si verifichi infatti quali sono le sequenze di azioni generate da un goal della forma `do(P,s0,S)`:

```
?- do(P,s0,_).
P = move(a,c) ;
P = move(c,a) ;
P = moveToTable(a) ;
P = (move(a,c):move(b,a)) ;
P = (move(a,c):move(a,b)) ;
P = (move(a,c):moveToTable(a)) ;
P = (move(a,c):move(b,a):moveToTable(b)) ;
P = (move(a,c):move(b,a):moveToTable(b):move(a,b)) ;
P = (move(a,c):move(b,a):moveToTable(b):move(b,a)) ;
P = (move(a,c):move(b,a):moveToTable(b):moveToTable(a)) ;
P = (move(a,c):move(b,a):moveToTable(b):move(a,b):move(c,a)) ;
P = (move(a,c):move(b,a):moveToTable(b):move(a,b):move(a,c)) ;
P = (move(a,c):move(b,a):moveToTable(b):move(a,b):moveToTable(a)) ;
..............
```

Poiché non vi sono "controlli di ciclo", la generazione di piani va avanti all'infinito senza mai generarne uno che risolve il problema dato.

È necessario dunque controllare la ricerca del piano. Nel seguito di questa sezione considereremo due modi diversi di controllare la ricerca in modo da garantirne la terminazione: in profondità limitata e in ampiezza.

3.7.1 Ricerca in profondità limitata

Il codice illustrato in questo paragrafo (ripreso sempre da [7]) implementa in Golog un semplice pianificatore in profondità, limitando la ricerca dei piani a quelli di una lunghezza fissata inizialmente. Ovviamente si assume che sia stato definito un dominio ed un predicato `goal(S)` che determina se `S` è una situazione obiettivo.

```
:- [golog_swi].
:- discontiguous proc/2.

%  Implementation of a simple depth-first planner.
% plandf(N): ricerca un piano di lunghezza massima N
%            la ricerca e' in profondita'
plandf(N) :- do(depthFirstPlanner(N),s0,_), askForMore.
askForMore :- write('More? '), read(n).

% procedura per la ricerca in profondita' limitata
proc(depthFirstPlanner(N),
```

```
?(goal) : ?(prettyPrintSituation) #
?(N > 0) : pi(a,?(primitive_action(a)) : a) :
    pi(n1, ?(n1 is N - 1) : depthFirstPlanner(n1))).
```

La procedura Golog `depthFirstPlanner(N)` verifica inizialmente se la situazione corrente è una situazione obiettivo. Il predicato `goal` è uno pseudo-fluente, da applicare alla situazione corrente, e richiede dunque la definizione di un'apposita clausola `restoreSitArg`:

```
restoreSitArg(goal,S,goal(S)).
```

Se la situazione corrente è un obiettivo, viene stampata (la definizione del predicato `prettyPrintSituation` è data in seguito). Altrimenti, se il limite di lunghezza `N` non è stato ancora raggiunto, si sceglie un'azione primitiva e si esegue; la procedura è poi richiamata ricorsivamente con limite `N-1`.

Quando il limite massimo viene raggiunto e la situazione corrente non è una situazione obiettivo, la procedura fallisce. Ciò determina l'esecuzione del backtracking sulla scelta delle azioni primitive da eseguire, fino a trovare una soluzione (se esiste).

Il goal Prolog principale, `plandf(N)`, invoca l'esecuzione della procedura Golog `depthFirstPlanner(N)`. Quando è stata trovata una soluzione, stampa il messaggio "`More?`" e, se l'utente immette un carattere diverso da `n`, fallisce. Il fallimento provoca una nuova esecuzione del backtracking che porterà alla ricerca di un'altra soluzione.

Il predicato `prettyPrintSituation` è trattato come uno pseudo-fluente (e in quanto tale ha bisogno della sua clausola `restoreSitArg`). È un fluente sempre vero, ma è definito in modo da avere come effetto collaterale la stampa della sequenza di azioni corrispondenti alla situazione suo argomento.

```
restoreSitArg(prettyPrintSituation,S,prettyPrintSituation(S)).

prettyPrintSituation(S) :- makeActionList(S,Alist), nl,
                           write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1),
                              append(L1, [A], L).
```

Come esempio di esecuzione, consideriamo il problema con stato iniziale e obiettivo definiti a pagina 59:

```
?- plandf(3).

[moveToTable(a), move(b, c), move(a, b)]
More? n.

?- plandf(6).

[move(a, c), move(b, a), moveToTable(b), moveToTable(a),
      move(b, c), move(a, b)]
More? y.
```

```
[move(a, c), move(a, b), move(a, c), moveToTable(a), move(b, c), move(a, b)]
More? n.
```

3.7.2 Ricerca in ampiezza

Il codice di questo paragrafo costituisce l'implementazione di un pianificatore in ampiezza. Per evitare che il programma entri in un ciclo infinito nel caso in cui non esistano soluzioni, viene comunque stabilito un limite massimo alla lunghezza dei piani. Il pianificatore implementa in effetti un meccanismo di ricerca in profondità per approfondimenti successivi (*iterative deepening*) del limite di lunghezza dei piani.

```
%   Implementation of World's Simplest Breadth-First Planner
planbf(N) :- do(breadthFirstPlanner(0,N),s0,_), askForMore.

% ricerca di un piano di lunghezza massima M, se M non supera N
proc(breadthFirstPlanner(M,N),
     ?(M =< N) :
     (actionSequence(M) : ?(goal) : ?(prettyPrintSituation) #
      pi(m1, ?(m1 is M + 1) : ?(reportLevel(m1))
             : breadthFirstPlanner(m1,N)))).

% esecuzione di una qualunque sequenza di azioni di lunghezza N:
% do(actionSequence(N),S,S') = S' e' il risultato dell'esecuzione
% a partire da S di una qualunque sequenza di azioni di lunghezza N
proc(actionSequence(N),
     ?(N = 0) #
     ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
       pi(n1, ?(n1 is N - 1) : actionSequence(n1))).

reportLevel(N) :- write('Starting level '), write(N), nl.
```

Il predicato Prolog principale, `planbf(N)` esegue la ricerca in ampiezza di un piano di lunghezza massima `N`. Esso invoca la procedura Golog `breadthFirstPlanner` con argomenti `0` e `N` ed in seguito la stessa procedura `askForMore` già vista nel paragrafo precedente.

La procedura `breadthFirstPlanner(M,N)` viene inizialmente invocata con `M=0`. Il parametro `M` viene poi incrementato ad ogni chiamata ricorsiva, finché non supera `N`.

L'esecuzione di `breadthFirstPlanner(M,N)`, quando $\texttt{M} \leq \texttt{N}$, genera (ed esegue) una sequenza di azioni primitive di lunghezza `M`; se la situazione corrente che si ottiene con l'esecuzione di tale sequenza di azioni è un obiettivo, la stampa. In alternativa, si richiama ricorsivamente incrementando `M`.

La procedura Golog `actionSequence(N)`, richiamata da `breadthFirstPlanner`, esegue una qualsiasi sequenza di azioni di lunghezza `N`:

```
?- do(actionSequence(2),s0,S).
S = do(move(b, a), do(move(a, c), s0)) ;
S = do(move(a, b), do(move(a, c), s0))
```

Ecco l'esecuzione del pianificatore sullo stesso esempio sopra considerato:

```
?- planbf(10).
Starting level 1
Starting level 2
Starting level 3

[moveToTable(a), move(b, c), move(a, b)]
More? y.
Starting level 4

[move(a, c), moveToTable(a), move(b, c), move(a, b)]
More? n.
```

3.7.3 Rappresentazione di conoscenza di controllo

Il pianificatore in ampiezza è decisamente meno efficiente del pianificatore in profondità. Tuttavia, la qualità dei piani trovati dal pianificatore in profondità è in generale molto scadente. Ad esempio:

```
?- plandf(10).

[move(a, c), move(b, a), moveToTable(b), move(a, b), move(c, a),
  moveToTable(c), move(a, c), moveToTable(a), move(b, c), move(a, b)]
```

Per problemi più complessi i tempi di esecuzione di una ricerca in avanti diventano presto molto elevati. Ad esempio, il pianificatore in profondità impiega più di 300 secondi per risolvere un problema con 6 blocchi (invocato con limite di lunghezza 12, su un PC con P4, 3.00GHz e 1GB RAM, sotto Linux).

Fin qui, però, non abbiamo sfruttato la ricchezza espressiva del linguaggio utilizzato per la rappresentazione del dominio. Negli approcci logici alla pianificazione è invece possibile sfruttare il potere espressivo del linguaggio per fornire una guida al pianificatore. In particolare è possibile fornire al pianificatore sia conoscenza specifica sul dominio, sia conoscenza di controllo (o conoscenza *euristica*), cioè informazioni che riducono il numero di piani accettabili, dunque lo spazio di ricerca del piano.

Nel caso dei pianificatori scritti in Golog, la conoscenza di controllo viene fornita sotto forma di un insieme di informazioni su "cattive situazioni" *bad situations*, cioè situazioni che non si devono (o è inutile) incontrare quando si genera un piano. Se durante la ricerca del piano si genera una "cattiva situazione", si deve abbandonare quella strada.

La definizione delle cattive situazioni è dipendente dal dominio. A volte inoltre è necessaria anche conoscenza specifica sul problema particolare. Ogni dominio, ed eventualmente ogni problema, dovrà dunque essere corredato dalla definizione di un predicato `badSituation(S)`, vero se `S` è una cattiva situazione.

La procedura di pianificazione in profondità limitata si può modificare in modo che abbandoni la ricerca non appena si genera una *bad situation*, come segue.

```
proc(depthFirstPlanner(N),
     ?(goal) : ?(prettyPrintSituation) #
     ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
         ?(-badSituation) :
```

```
        pi(n1, ?(n1 is N - 1) : depthFirstPlanner(n1))).
```

Analogamente, per la ricerca in ampiezza, si può modificare la procedura `actionSequence` in modo che vengano generate soltanto sequenze di azioni di lunghezza `N` che non risultino in una *bad situation*:

```
proc(actionSequence(N),
    ?(N = 0) #
    ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
      ?(-badSituation) :
      pi(n1, ?(n1 is N - 1) : actionSequence(n1))).
```

Si noti che i pianificatori controllano la verità di `badSituation` sulla situazione corrente. Tale predicato è da trattare come un fluente, ed è dunque necessario inserire nel programma un'apposita clausola `restoreSitArg`:

```
restoreSitArg(badSituation,S,badSituation(S)).
```

La definizione delle cattive situazioni per ciascun dominio e problema non è un compito facile. Nel caso del mondo dei blocchi, [7] propone la definizione delle cattive situazioni in termini di "buone torri", cioè quelle che non dovranno mai essere smontate per raggiungere il goal. La definizione delle buone torri dipende dal goal, quindi dal problema particolare che si deve risolvere.

Ad esempio, se l'obiettivo è quello di costruire la torre con `a` sopra `b`, `b` sopra `c` e `c` sul tavolo:

```
goal(S) :- ontable(c,S), on(a,b,S), on(b,c,S), clear(a,S).
```

si possono definire le buone torri come segue:

```
%%% goodTower(X,S) = nella situazione S c'e' una buona torre
%                    con X in cima
goodTower(a,S) :- on(a,b,S), goodTower(b,S).
goodTower(b,S) :- on(b,c,S), goodTower(c,S).
goodTower(c,S) :- ontable(c,S).
```

Le *bad situations* nel mondo dei blocchi si possono allora identificare come segue:

1. Se si sposta un blocco sopra un altro creando una torre che non è buona, allora si ottiene una cattiva situazione:

   ```
   %  Don't create a bad tower by a move action.
   badSituation(do(move(X,Y),S)) :- not(goodTower(X,do(move(X,Y),S))).
   ```

2. Se si distrugge una buona torre, allora si ottiene una cattiva situazione:

   ```
   %  Don't move anything from a good tower to the table.
   badSituation(do(moveToTable(X),S)) :- goodTower(X,S).
   ```

3. Se si ha l'opportunità di creare una buona torre, e invece si costruisce una torre che non è buona, allora si ottiene una cattiva situazione:

```
/*  Opportunistic rule: If an action can create a good tower, don't
    do a bad-tower-creating moveToTable action.  */
badSituation(do(moveToTable(X),S)) :-
                            not(goodTower(X,do(moveToTable(X),S))),
                            existsActionThatCreatesGoodTower(S).

existsActionThatCreatesGoodTower(S) :-
                        (A = move(Y,_) ; A = moveToTable(Y)),
                        poss(A,S), goodTower(Y,do(A,S)).
```

Concludiamo questo paragrafo con alcune osservazioni sull'efficienza dei pianificatori Golog. Il pianificatore in ampiezza genera sempre piani ottimi (di lunghezza minima), ma è piuttosto inefficiente relativamente ai tempi di esecuzione, anche quando si include conoscenza di controllo.

Al contrario, il pianificatore in profondità, quando gli è fornita una buona conoscenza di controllo, è piuttosto efficiente, sia rispetto al tempo di esecuzione che alla qualità dei piani. Tuttavia, si deve osservare che se il limite di profondità con il quale viene invocato non è sufficiente a trovare una soluzione, può impiegare molto tempo per scoprirlo.

3.7.4 Esercizi

Considerare i seguenti domini di pianificazione e rappresentarli in Prolog in modo che siano utilizzabili dai pianificatori Golog. Per ciascun dominio, definire anche un'opportuna conoscenza di controllo che aiuti il pianficatore. Nella descrizione che segue, per ciascun dominio è definita un'istanza di problema; definirne comunque anche altre e provare il comportamento dei pianificatori sulle diverse istanze.

1. **Dominio:** una scimmia è in una stanza, dove delle banane sono appese al soffitto, fuori della sua portata. Nella stanza c'è anche un panchetto, sul quale la scimmia potrebbe salire per raggiungere le banane. Le azioni possibili per la scimmia sono: andare da un posto all'altro nella stanza, spingere un oggetto da un posto all'altro, salire su un oggetto e afferrare un oggetto.

   Problema: le posizioni della stanza sono A, B, C e D; inizialmente, la scimmia è in A, le banane sono appese in B e il panchetto è in C. Gli altri oggetti presenti nella stanza sono una palla, inizialmente in D, e un libro, inizialmente in B. L'obiettivo è quello di avere le banane.

2. **Dominio:** Un robot con un solo braccio agisce in un ambiente costituito da una sola stanza e un corridoio. Le azioni che può compiere sono entrare e uscire dalla stanza, aprire borse, chiudere borse, mettere oggetti piccoli dentro borse (solo se queste sono aperte), prendere in mano borse (solo se queste sono chiuse).

   Problema: nello stato iniziale il robot è fuori della stanza. Nella stanza ci sono un tavolo (grande), una borsa chiusa e un libro (piccolo) appoggiato sul tavolo. L'obiettivo del robot è quello di trovarsi fuori della stanza, con il libro dentro la borsa.

3. **Dominio:** 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.

   Problema: i posti dell'ambiente sono *casa*, *super* (negozio di alimentari) e *hs* (hardware store, che vende attrezzi da bricolage). Inizialmente il robot è a casa e non possiede alcun oggetto. L'obiettivo è quello di essere a casa con un litro di latte, una banana e un trapano.

4. **Dominio (Gripper):** Un robot con due braccia agisce in un ambiente costituito da due stanze, A e B. Ogni braccio del robot può tenere un solo oggetto alla volta. Le azioni che il robot può compiere sono "prendere la palla X con la mano Y nella stanza Z", "lasciare la palla X tenuta con la mano Y nella stanza X", andare dalla stanza X alla stanza Y".

   Problema: Inizialmente la stanza A contiene 4 palle. L'obiettivo è quello di avere le 4 palle nella stanza B.

5. **Dominio (Briefcase):** 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.

   Problema: nell'ambiente ci sono 4 locazioni (*casa*, A. B, C) e 3 oggetti. Inizialmente il robot è a *casa*, l'oggetto 1 è in A, l'oggetto 2 è in B e l'oggetto 3 è in C. L'obiettivo è quello di avere tutti e 3 gli oggetti a *casa* (fuori della borsa).

3.8 Conclusioni

Il Golog è un 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).

Tra le sue caratteristiche principali, ricordiamo:

- 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 di tipo 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 eseguite. È compito del theorem prover generare le sequenze di azioni primitive direttamente eseguibili dall'esecutore.

Negli ultimi anni sono state proposte diverse estensioni di Golog al fine di rendere il linguaggio applicabile a domini realistici. In particolare, Golog è stato esteso in modo da includere azioni concorrenti (ConGolog), un trattamento esplicito del tempo, la modellazione di cambiamenti continui, e azioni di sensing.

Una versione di ConGolog che permette la trattazione del tempo e cambiamenti continui è attualmente usata per controllare i soccer robots delle Università di Toronto(gruppo di Robotica Cognitiva del prof. Levesque) e di Aachen (gruppo di Robotica Cognitiva del prof. Lakemeyer) che giocano nella mid-size league.

Bibliografia

[1] Cognitive Robotics Group; Università di Toronto. `http://www.cs.toronto.edu/cogrobo/`.

[2] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[3] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[4] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[5] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[6] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[7] R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, 2001. `http://www.cs.toronto.edu/cogrobo/kia/index.html`.