DLV

Nicola Gatta
Matr. 35042

2 dicembre 2001

1 Introduzione a DLV

DLV è un sistema di deduzione basato su database e programmazione logica disgiuntiva creato da un team italo-austriaco guidato da *Nicola Leone* (leone@unical.it del Dipartimento di Matematica dell'università della Calabria.

2 Primo approccio e uso di DLV

DLV è distribuito in forma binaria per vari sistemi operativi e, nella versione per Linux, non ha un'interfaccia grafica. Funziona dunque in riga di comando.
Un tipico utilizzo di *DLV* è il seguente:

```
$ dlv FILE [FILE]
```

in cui il (o i) file di input contiene (contengono) la descrizione del dominio e dei fatti.
Ad esempio si potrebbe avere un file contenente:

```
mangia(X) :- fame(X).
fame(pippo).
```

La prima regole (d'ora in poi si riferirà ad una come *rule*) modellizza il fatto che quuando una persona X ha fame allora mangia. La seconda invece afferma che pippo ha fame. Infatti *DLV* darà come output le seguenti righe:

```
dlv [build BEN/Jun 11 2001 gcc 2.95.2 19991024 (relea-
se)]
{fame(pippo), mangia(pippo)}
```

L'output sta ad indicare che dal fatto di base *fame(pippo)* DLV ha inferito che pippo mangia.

DLV ha molte funzionalità che espandono la programmazione logica, adattandola a vari contesti applicativi, tra cui la pianificazione, come viene mostrato in appendice.

3 Il linguaggio *Disjunctive Datalog*

Nel primissimo esempio sono stati mostrati costrutti del linguaggio che DLV mette a disposizione del progettista di un dominio per la descrizione del dominio. Ora si vedranno i rudimenti di tale linguaggio che in gergo tecnico è detto **Datalog with Disjunction**.

3.1 DLV e linguaggio Datalog with disjunction

Come visto nel primo esempio DLV riceveva in input due clausole intrinsecamente diverse: la clausola `mangia(X) :- fame(X).` è generale (esprime un fatto vero in qualunque situazione); la seconda clausola `fame(pippo).` esprime il fatto che pippo ha fame .

In altre parole la seconda clausola è un'**istanza**, nel senso che è una particolarizzazione del fatto *fame(X)*.

In genere si preferisce separare i due tipi di clausola riferendosi ad essi come:

- **Intensional Database**: insieme di rule di tipo generale

- **Estensional Database**: insieme dei fatti.

Volendo fare un parallelo con la programmazione si potrebbe dire che

- al **Intensional Database** corrisponde il codice che implementa un algoritmo

- al **Estensional Database** corrispondono le strutture dati inizializzate.

NOTA: ogni rule o fatto, in DLV deve essere terminata dal carattere "." (punto).

3.2 Commenti

In un file che descrive un dominio si possono includere dei commenti: tutte le righe del file che iniziano per ”%” vengono scartate e non considerate dal DLV.

4 Le RULES del linguaggio

La sintassi del linguaggio ”Datalog with Disjunction” è mutuata dal `Prolog`.

DLV usa una rappresentazione delle clausole basate su logica del primo ordine

DLV accetta nelle regole l'uso di variabili, distinguendo:

- le variabili devono iniziare con una lettera maiuscola

- qualsiasi cosa che non è una variabile deve iniziare con una lettera minuscola o un numero.

4.1 Implicazioni

Le implicazioni vengono tradotte come in Prolog, cioè facendo precedere l'*head* dal *body*:

```
mangia(X) :- fame(X).
```

sta ad indicare la seguente clausola del primo ordine:

$$\forall x \qquad fame(x) \Longrightarrow mangia(x)$$

Nel head di una implicazione è possibile esprimere una congiunzione di fatti nel seguente modo:

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

sta ad indicare la seguente clausola del primo ordine:

$$\forall x, y, z \qquad genitore(x, y) \wedge antenato(y, z) \Longrightarrow antenato(x, z)$$

DLV non permette di esprimere l'uso di quantificatori esistenziali, anche se, in genere, si può trasformare sempre un quantificatore esitenziale in un quantificatore universale.

Una nota positiva di DLV è la capacità di trattare regole che presentano una "disgiunzione" nel body dell'implicazione

$$\forall x \qquad persona(x) \Longrightarrow maschio(x) \vee femmina(x)$$

può essere tradotta come

```
maschio(X) v femmina(X) :- persona(X).
```

4.2 Negazione

In logica del primo d'ordine si indica con "atomo" un'espressione del tipo
`genitore(x,y)`,
con il quale si indica che "x è genitore di y" DLV quando incontra per la prima volta un atomo, non ne conosce lo stato di verità e quindi lo considera falso; quando trova una regola (o meglio ancora un fatto) che specifichi il suo stato allora si adeguerà. Questa assunzione (detta *CWA = Complete World Assumption*) però fa sorgere un problema: quando si incontri un atomo di cui non si sa ancora nulla ed esso è fondamentale per un'implicazione cosa si fa?

Esempio
Si ha

```
sentebarzelletta(X) :- barzelletta(X), not sordo(X).
barzelletta(pippo).
```

A pippo viene detta una barzelletta: se pippo non è sordo allora la sente. Il problema è che nel dominio non viene disambiguato se pippo è sordo oppure no: secondo la CWA sordo(X) sarebbe falso, e quindi piipo sente la barzelletta.

In alcuni casi è però sbagliato fare un'assunzione di questo tipo, quindi DLV permette di distinguere due tipi di negazione nelle sue rule:

- Negazione debole indicata con "**not**"; l'atomo *not p(X)* risulta vero anche se di p(X) non si sa nulla (si segue la CWA)

- Negazione forte indicata con "**-**"; si forza a controllare il valore dell'atomo. L'atomo *-p(X)* risulta vero se e solo se la falsità di p(X) è già stata in qualche modo dimostrata, in caso contrario tutte le clausole dipendenti da p(X) vengono "inibite".

4.3 Vincoli

I vincoli sono tutte quelle espressioni che in qualche pongono dei limiti alle regole implicative. In generale le implicazioni, dati dei fatti iniziali, mirano a dimostrare dei nuovi fatti che vanno ad accumulare la Knowledge Base. I vincoli invece regolano il comportamento delle implicazioni. Naturalmente se il dominio e' ben espresso mediante implicazioni, i vincoli non sono necessari; d'altra parte è in genere impossibile esprimere un dominio di una qualche utilità senza l'uso di vincoli. Già per problemi "giocattolo" come il mondo dei blocchi per la pianificazione è molto conveniente esprimere il dominio con l'aiuto dei vincoli. In DLV è possibile esprimere due tipi di vincoli: vincoli normali e vincoli deboli (qusti ultimi detti anche vincoli con priorità e peso)

4.3.1 Vincoli Normali

Vengono espressi come congiunzione di fatti che non possono verificarsi in una stessa istanza di un dominio. Un tipico esempio è:

$$\forall x \qquad maschio(x) \Longrightarrow \neg femmina(x)$$

un maschio non può essere femmina. In DLV si esprime con

```
:- maschio(X), femmina(X).
```

affermando che un qualsiasi X non può essere contemporaneamente maschio e femmina.

4.3.2 Vincoli Deboli

Sono una delle funzionalità più avanzate di DLV. I vincoli deboli sono dei vincoli che, in determinate situazioni, possono essere violati. La sintassi è:

```
:~ vincolo.  [Peso:Priorità]
```

in cui Peso e Priorità sono degli interi positivi facoltativi.
A seconda dei valori di Peso e Priorità DLV si comporta in diversi modi:

- Peso e Priorità non specificati: DLV sceglie il modello (insieme di inferenze) che violano il minor numero possibile di vincoli deboli.
- Peso indicato e Priorità costante per tutti i vincoli: viene data una valutazione ad un fatto vincolato. Tra più fatti in concorrenza fra loro viene scelto quello a peso inferiore.

- Peso e Priorità indicato: si raggruppano i vincoli in livello distinti fra loro. In questo caso ci può essere anche più di un unico modello che soddisfa i vincoli

5 Costrutti aggiuntivi

5.1 Predicati comparativi

I confronti sono quelli messi a disposizione da tutti i linguaggi di porogrammazione

minoranza	<
maggioranza	>
uguaglianza	==
disuguaglianza	!=
minoranza o uguaglianza	<=
maggioranza o uguaglianza	>=

Si noti come essi siano applicabili a qualsiasi simbolo, anche se l'utilizzo in ambiti non numerici dà risultati mediante la comparazione delle costanti viste come stringhe.

5.2 Predicati aritmetici

Sono solo quattro:

somma	+
prodotto	*
controllo se intero	#int(X)
controllo se successore	#succ(X,Y)

Esempi

+(X,Y,Z) restituisce true se X=Y+Z,
**(X,Y,Z)* restituisce true se X=Y*Z,
#int(X) restituisce true se X è un intero
#succ(X,Y) restituisce true se Y=X+1

6 Caso di studio: Pianificazione nel mondo della scimmia

In questo dominio si hanno quattro stanze e quattro oggetti: una scimmia, una sedia, un coltello e una banana appesa con filo al soffitto. Le stanze comunicano tra di loro direttamente, nel senso che è possibile passare senza problemi da una all'altra senza percorsi intermedi. La scimmia, prima di poter mangiare la banana deve posizionare la sedia sotto la banana, salirci sopra e tagliare la corda che lega la banana al soffitto. Gli oggetti si trovano inizialmente in stanze diverse. Lo scopo è trovare un piano che in un certo numero di mosse permetta alla scimmietta di mangiare la banana.
<u>Nota</u>: nella pianificazione in DLV è necessario tenere conto del passare del tempo mediante degli slot di tempo interi: per fare questo è necessario passare al DLV un'opzione su quanti slot di tempo compongono il piano (ammettendo che ad ogni slot di tempo si può eseguire solo un'azione).
Ecco il listato completo del dominio con relativi commenti:

```
%prima regola: in un dato istante di tempo la scimmia puo' scegliere se
% muoversi ->move(time)
% spostare la sedia ->move_chair(time)
% salire sulla sedia -> ascend(time)
% prendere il coltello -> getknife(time)
% non fare nulla -> idle(time)

move(Time) v move_chair(Time) v ascend(Time) v getknife(Time) v idle(Time)
:- #int(Time).

% seconda e terza regola: se la scimmia si muove o sposta la sedia allora
% e' in movimento:

monkey_motion(T) :- move(T).
monkey_motion(T) :- move_chair(T).

% quarta regola: se la scimmia sale sulla sedia allora al passo successivo
% si trova sulla sedia

onbox(monkey, T2) :- ascend(T) , T2=T+1.

% quinta regola: se la scimmia afferra il coltello allora dall'istante
% successivo possiede il coltello.
```

hasknife(T2) :- getknife(T1), T2=T1+1.

% sesta, settima,ottava e nona regola: se la scimmia sta sulla sedia,
% non puo' ne' salire sulla sedia (e' gia' sopra), ne' essere in
% movimento. Inoltre al passo successivo rimane sulla sedia
% (in pratica non puo' scendere). Inoltre se la scimmia possiede il
%coltello all'istante T allora all'istante T+1 lo possiede ancora
%(in pratica non puo' buttarlo via o lasciarlo)

```
:- onbox(monkey,T), ascend(T).
:- onbox(monkey,T), monkey_motion(T).
onbox(monkey,T2) :- onbox(monkey,T1), T2=T1+1.
hasknife(T2) :- hasknife(T1), T2=T1+1.
```

% decima regola: se al passo T la sedia non viene mossa allora la sua
% posizione rimane la stessa al passo T+1

```
at(chair,X,T2) :- at(chair,X,T1), T2=T1+1, not move_chair(T1).
```

%undicesima regola: se il coltello non e' in mano alla scimmia rimane
% dove era originariamente

```
at(knife,X,T2) :- at(knife,X,T1), T2=T1+1, not hasknife(T1).
```

% dodicesima e tredicesima regola: come viene spostata la sedia e il
% coltello. Infatti se la scimmia muove
% la sedia al passo T1, allora la sedia al passo T2=T1+1 si trovera' nello
% stesso posto in cui si trova la scimmia al passo T2=T1+1.
% E' giusta ma decisamente poco naturale come formalizzazione
% Lo stesso vale per il coltello, ammesso che la scimmia lo possieda

```
at(knife, Pos2, T2) :- hasknife(T1), T2=T1+1, at(monkey,Pos2,T2).
at(chair, Pos2, T2) :- move_chair(T1), T2=T1+1, at(monkey,Pos2,T2).
```

% quattordicesima regola: ci sono tre stanze distinte, una per la banana,
% un'altra e' la stanza in cui la sedia si trova all'inizio, la terza e'
% quella in cui la scimmia si trova all'inizio, l'ultima e' del coltello.
% queste tre stanze sono formalizzate considerando il fatto che la scimmia
% essere solo in una di queste (disgiunzione) e non in altri posti (anche
% questa e' poco naturale).

```
at(monkey,monkey_starting_point,T) v at(monkey,chair_starting_point,T) v
at(monkey,below_banana,T) v at(monkey,knife_starting_point,T) :- #int(T).

% quindicesima regola: la scimmia non puo' essersi spostata senza aver fatto
% un movimento.

:- at(monkey,Pos1,T2), at(monkey,Pos2,T1), T2=T1+1, Pos1!=Pos2,
-monkey_motion(T1).

% sedicesima regola: se la scimmia si muove non puo' essere nella stessa
% posizione in due istanti consecutivi:

:- at(monkey,Pos,T2), at(monkey,Pos,T1), T2=T1+1, monkey_motion(T1).

% 17° regola: la scimmia non puo' salire sulla sedia se scimmia e
% sedia si trovano in luoghi differenti al momento della salita.

:- ascend(T), at(monkey,Pos1,T), at(chair,Pos2,T), Pos1<>Pos2.

% 18° e 19° regola: la scimmia non puo' afferrare il coltello se
% non si trova nello stesso posto del coltello e se ha gia' un
%coltello non puo' afferrarne un altro.

:- getknife(T), at(monkey,Pos1,T), at(knife,Pos2,T), Pos1<>Pos2.
:- getknife(T), hasknife(T).

% ventesima regola: la scimmia non puo' muovere la sedia s non si
% trova nella stessa stanza della sedia (= non ha accesso alla sedia)

:- move_chair(T), at(monkey,Pos1,T), at(chair,Pos2,T), Pos1<>Pos2.

% ventunesima regola(aggiunta forzata per far funzionar la modifica
% con coltello): la scimmia non puo' muoversi se sta sulla sedia

:- move(T), onbox(monkey,T).

% ventiduesima regola : se la scimmia e' sulla sedia e la sedia e' nella
% stessa stanza della banana ed ha il coltello allora puo' mangiare.
```

mangia :- onbox(monkey,T), at(chair,below_banana,T),
at(monkey,below_banana,T), hasknife(T).

% A questo punto manca solo la parte estensionale, ovvero i fatti iniziali:
% la scimmia , la sedia e la banana, al'istante iniziale si trovano in tre
% stanze diverse (lo si fa mettendo scimmia sedia coltello in posti diversi
% che non contengono banana, visto che non c'e' necessita' di muovere
% la banana e quindi non c'e' l'operatore banana_at_place(X,T).

at(monkey,monkey_starting_point,0).
at(chair,chair_starting_point, 0).
at(knife,knife_starting_point, 0).

% le seguenti regole servono SOLO a visualizzare le azioni in modo
% user-friendly

azione(N, vai_a, Destinazione) :- move(N), at(monkey,Destinazione, N2), N2=N+1.
azione(N, sposta_sedia_a, Destinazione) :- move_chair(N), at(chair,Destinazione, N2), N2=N+1.
azione(N, sali,:-)) :- ascend(N).
azione(N, prendi_coltello,) :- getknife(N).

% ed il goal:

:- not mangia.

L'output di DLV, opportunamente organizzato per una maggiore leggibilità, è stato il seguente:

dlv [build BEN/Jun 11 2001 gcc 2.95.2 19991024 (release)]

{at(monkey,monkey_starting_point,0),
at(chair,chair_starting_point,0),
at(knife,knife_starting_point,0),
move(0),
monkey_motion(0),
azione(0,vai_a,knife_starting_point),

at(knife,knife_starting_point,1),
at(monkey,knife_starting_point,1),

at(chair,chair_starting_point,1),
getknife(1),
azione(1,prendi_coltello,),

hasknife(2),
at(monkey,knife_starting_point,2),
at(knife,knife_starting_point,2),
at(chair,chair_starting_point,2),
move(2),
monkey_motion(2),
azione(2,vai_a,chair_starting_point),

hasknife(3),
at(monkey,chair_starting_point,3),
at(knife,chair_starting_point,3),
at(chair,chair_starting_point,3),
move_chair(3),
monkey_motion(3),
azione(3,sposta_sedia_a,below_banana),

hasknife(4),
at(monkey,below_banana,4),
at(knife,below_banana,4),
at(chair,below_banana,4),
ascend(4),
azione(4,sali,“),

hasknife(5),
at(monkey,below_banana,5),
at(knife,below_banana,5),
at(chair,below_banana,5),
onbox(monkey,5),
idle(5),

mangia}

Nota: In questo output i numeri da 1 a 5 indicano gli istanti di tempo: per le azioni si intende l’istante in cui esse vengono applicate, per i predicati si intende l’istante in cui essi sono validi.