Constraint Programming & Planning

Agostino Dovier

Università di Udine
Dipartimento di Matematica e Informatica

Udine, Maggio 2011

SOMMARIO

Answer Set Programming

- Answer Set Programming, in breve, ASP, è una metodologia di programmazione che usa come linguaggio un sottoinsieme di prolog, con negazione e restrizioni sintattiche sull'uso dei simboli funzionali, in modo da assicurare programmi ground finiti
- Il codice viene scritto avendo in mente la semantica del modello stabile.
- Applicazioni: Knowledge Representation, Planning e Combinatorial Problems.

REGOLE, PROGRAMMI, E CODICI ASP

- Una *regola (ASP)* è una clausola della forma:

$$L_0 \leftarrow L_1, \ldots, L_m, \texttt{not}\ \ L_{m+1}, \ldots, \texttt{not}\ \ L_n.$$

  dove ogni L_i è una formula atomica.
- Un *vincolo (ASP)* è una clausola della forma:

$$\leftarrow L_1, \ldots, L_m, \texttt{not}\ \ L_{m+1}, \ldots, \texttt{not}\ \ L_n.$$

  la cui semantica logica è la disgiunzione:
  $\neg L_1 \vee \cdots \vee \neg L_m \vee L_{m+1} \vee \cdots \vee L_n$.
- Un *programma* ASP è un insieme di regole ASP.
- Un *codice* ASP è un insieme di regole e vincoli ASP.
- Un programma ASP ammette sempre un modello, un codice no (anche se ...)

REGOLE, PROGRAMMI, E CODICI ASP

- E' un formalismo basato sulla semantica: l'ordine dei letterali non deve avere alcuna importanza.
- In Prolog l'esecuzione di un goal avviene in modo *top-down* e *goal-directed*.
- In ASP si opera *bottom-up*. Ciò non fa cadere ASP in loop banali (p.es. $p \leftarrow p$).
- Il costrutto extra-logico CUT non è presente in ASP (al massimo c'è un po' di aritmetica e i vincoli di cardinalità)
- I programmi logici trattabili con gli attuali ASP-solvers devono rispettare delle restrizioni sull'impiego delle variabili.

Esempio KR

Vogliamo modellare il fatto che gli uccelli volino e al contempo che i pinguini, che sono uccelli, non siano in grado di farlo.

```
vola(X) :- uccello(X), not anormale(X).
anormale(X) :- pinguino(X).
uccello(X) :- pinguino(X).
uccello(titti).
pinguino(pingu).
```

ESEMPIO KR

Costruiamo l'istanza ground:

```
vola(titti) :- uccello(titti), not anormale(titti).
vola(pingu) :- uccello(pingu), not anormale(pingu).
anormale(titti) :- pinguino(titti).
anormale(pingu) :- pinguino(pingu).
uccello(titti) :- pinguino(titti).
uccello(pingu) :- pinguino(pingu).
uccello(titti).
pinguino(pingu).
```

`uccello(titti)` e `pinguino(pingu)` devono appartenere a qualsiasi answer set

Di conseguenza anche `uccello(pingu)` e `anormale(pingu)`

Aggiungendo `vola(titti)` otteniamo l'answer set

ESEMPIO KR

Costruiamo l'istanza ground:

```
vola(titti) :- uccello(titti), not anormale(titti).
vola(pingu) :- uccello(pingu), not anormale(pingu).
anormale(titti) :- pinguino(titti).
anormale(pingu) :- pinguino(pingu).
uccello(titti) :- pinguino(titti).
uccello(pingu) :- pinguino(pingu).
uccello(titti).
pinguino(pingu).
```

`uccello(titti)` e `pinguino(pingu)` devono appartenere a qualsiasi answer set

Di conseguenza anche `uccello(pingu)` e `anormale(pingu)`

Aggiungendo `vola(titti)` otteniamo l'answer set

ESEMPIO KR

Costruiamo l'istanza ground:

```
vola(titti) :- uccello(titti), not anormale(titti).
vola(pingu) :- uccello(pingu), not anormale(pingu).
anormale(titti) :- pinguino(titti).
anormale(pingu) :- pinguino(pingu).
uccello(titti) :- pinguino(titti).
uccello(pingu) :- pinguino(pingu).
uccello(titti).
pinguino(pingu).
```

`uccello(titti)` e `pinguino(pingu)` devono appartenere a qualsiasi answer set
Di conseguenza anche `uccello(pingu)` e `anormale(pingu)`
Aggiungendo `vola(titti)` otteniamo l'answer set

ESEMPIO KR

Costruiamo l'istanza ground:

```
vola(titti) :- uccello(titti), not anormale(titti).
vola(pingu) :- uccello(pingu), not anormale(pingu).
anormale(titti) :- pinguino(titti).
anormale(pingu) :- pinguino(pingu).
uccello(titti) :- pinguino(titti).
uccello(pingu) :- pinguino(pingu).
uccello(titti).
pinguino(pingu).
```

`uccello(titti)` e `pinguino(pingu)` devono appartenere a qualsiasi answer set
Di conseguenza anche `uccello(pingu)` e `anormale(pingu)`
Aggiungendo `vola(titti)` otteniamo l'answer set

THEOREM

Dato un programma P, ogni answer set di P è un modello minimale di P.

THEOREM

Sia P un programma e sia S un insieme di atomi. Allora S è answer set di P se e solo se

- *S è modello di P; e*
- *per ogni S', se S' è modello di P^S allora $S \subseteq S'$.*

THEOREM

Il problema di stabilire se un programma generale ground P ammette modelli stabili è NP-completo.

- I vincoli sono regole prive di testa:

$$\leftarrow L_1, \ldots, L_m, \texttt{not}\ \ L_{m+1}, \ldots, \texttt{not}\ \ L_n.$$

- Una tale regola ha l'effetto di invalidare ogni answer set che ne soddisfi il corpo, ovvero ogni insieme di atomi che contenga tutti gli atomi $L_1, \ldots, L_m$ e nessuno degli $L_{m+1}, \ldots, L_n$.

I vincoli sono una estensione comoda ma puramente sintattica.

- I vincoli sono regole prive di testa:

$$\leftarrow L_1, \ldots, L_m, \texttt{not}\ \ L_{m+1}, \ldots, \texttt{not}\ \ L_n.$$

- Una tale regola ha l'effetto di invalidare ogni answer set che ne soddisfi il corpo, ovvero ogni insieme di atomi che contenga tutti gli atomi $L_1, \ldots, L_m$ e nessuno degli $L_{m+1}, \ldots, L_n$.

I vincoli sono una estensione comoda ma puramente sintattica.

Consideriamo il seguente programma

```
a :- not n_a.   n_a :- not a.
b :- not n_b.   n_b :- not b.
c :- not n_c.   n_c :- not c.
d :- not n_d.   n_d :- not d.
```

Gli answer set di questo programma rappresentano tutti i modi possibili di scegliere una e una sola alternativa tra

- `a` e `n_a`,
- `b` e `n_b`,
- `c` e `n_c`,
- `d` e `n_d`.

```
scelto(X) :- possibile(X), not non_scelto(X).
non_scelto(X) :- possibile(X), not scelto(X).
p :- possibile(X), scelto(X).
:- not p.
possibile(a).  possibile(b).  possibile(c).
```

Answer sets (S = {p, possibile(a), possibile(b), possibile(c)}):

S ∪ {scelto(a), scelto(b), scelto(c)}
S ∪ {scelto(a), scelto(b), non_scelto(c)}
....
S ∪ {scelto(b), non_scelto(a), non_scelto(c)}
S ∪ {scelto(c), non_scelto(a), non_scelto(b)}

```
diff_scelto(X) :- possibile(X),possibile(Y),
                  scelto(Y), X!=Y.
scelto(X) :- possibile(X), not diff_scelto(X).
possibile(a).  possibile(b).  possibile(c).
```

Gli answer set sono:

$S \cup$ {scelto(a), diff_scelto(b), diff_scelto(c)}
$S \cup$ {scelto(b), diff_scelto(a), diff_scelto(c)}
$S \cup$ {scelto(c), diff_scelto(a), diff_scelto(b)}

dove S ={possibile(a), possibile(b), possibile(c)}.

```
dominio(a).  dominio(b).  dominio(c).
codominio(0).  codominio(1).
fun(X,Y):- dominio(X),
     codominio(Y), codominio(Y1),
     Y != Y1, not fun(X,Y1).
```

Troviamo i modelli

1. `fun(a,0), fun(b,0), fun(c,0).`

...

8. `fun(a,1), fun(b,1), fun(c,1).`

Esercizio: farlo con codominio non booleano.

ASP SOLVERS

LPARSE+SMODELS

- Libero, sviluppato presso la Helsinki University of Technology.
- Composto da due strumenti: lparse e smodels.
- lparse funge da front-end ed effettua il *grounding*.
- L'output di lparse viene poi processato da smodels che calcola effettivamente gli answer sets del programma grounded (e quindi di P).

ASP SOLVERS

LPARSE

- Per rendere finito il processo di grounding e generare un programma *ground*(P) composto da un numero finito di regole, il programma P deve rispettare alcuni vincoli.
- I simboli di funzione *non sono vietati*, ma è consentito solamente un uso limitato.
- Il programma deve essere *strongly range restricted*.
- L'idea base è che il programma P deve essere strutturato in modo che sia possibile, per ogni variabile di una regola, stabilire l'insieme di valori che essa può assumere
- tale insieme deve essere inoltre finito.

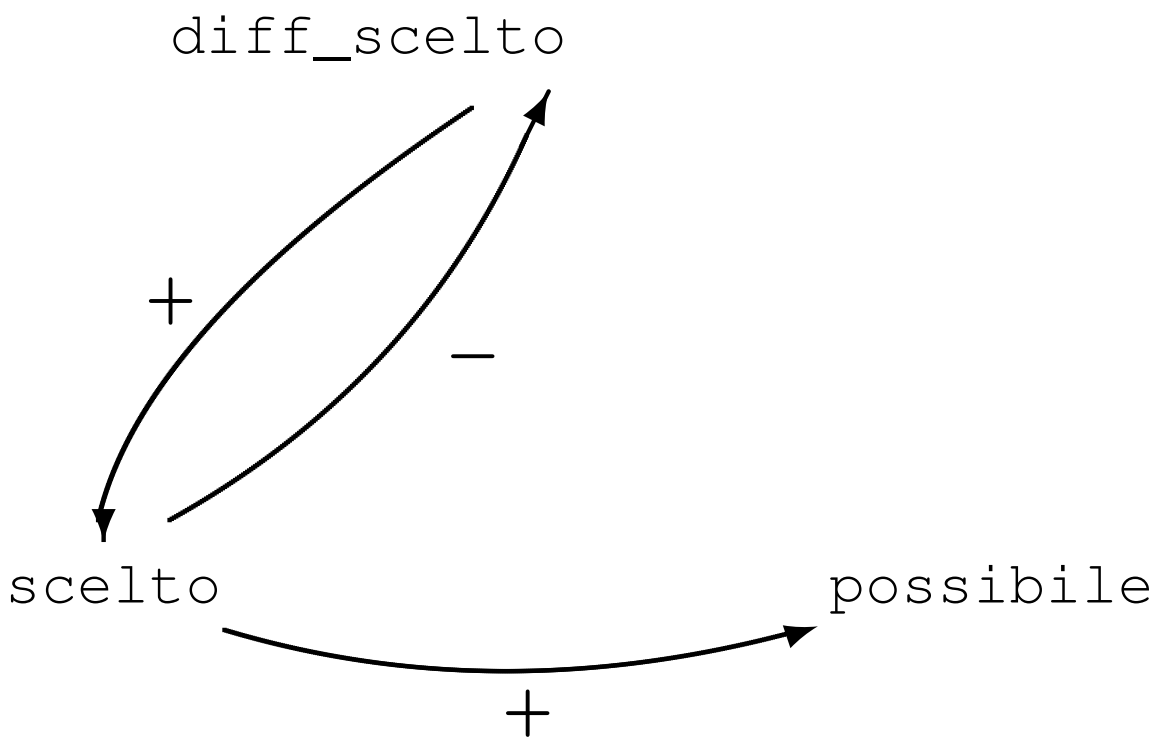
- Per definire strongly range restricted, si introduce la nozione di *dependency graph* di un programma ASP.
- Sia P un programma. Il *dependency graph* D_P di P è un grafo etichettato così definito:
- i nodi di D_P corrispondono ai simboli di predicato presenti in P;
- vi è un arco $\langle p_i, p_j, \ell \rangle$ tra i due nodi p_i e p_j se in P esiste una regola in cui p_i occorre nella testa e p_j occorre nel corpo.
- ℓ può essere uno o entrambi i simboli $+$, $-$ a seconda che il simbolo p_j occorra in un letterale positivo o negativo nel corpo della regola.
- Un ciclo nel grafo D_P si dice *ciclo negativo* se almeno una delle sue etichette contiene $-$.

- Un predicato p che occorre in P si dice *predicato di dominio* se e solo se in D_P ogni cammino che parte dal nodo p non contiene cicli negativi.
- Una regola ρ si dice *strongly range restricted* se ogni variabile che occorre in ρ occorre anche negli argomenti di un predicato di dominio nel corpo di ρ.
- Un programma P è *strongly range restricted* se tutte le sue regole sono strongly range restricted.

ASP SOLVERS

LPARSE

```
diff_scelto(X) :- scelto(Y), X!=Y.
scelto(X) :- possibile(X), not diff_scelto(X).
possibile(a).  possibile(b).  possibile(c).
```



Un Ground cardinality constraint si usa come un atomo positivo della forma

$$n\{\mathtt{L}_1,\ldots,\mathtt{L}_h,\ \mathtt{not}\ \mathtt{H}_1,\ldots,\ \mathtt{not}\ \mathtt{H}_k\}m$$

dove $\mathtt{L}_1,\ldots,\mathtt{L}_h,\mathtt{H}_1,\ldots,\mathtt{H}_k$ sono atomi e n e m sono numeri interi (uno o entrambi possono essere assenti). Definiamo, relativamente ad un insieme di atomi S e ad un cardinality constraint C il valore $val(C,S)$ come

$$val(C,S) = \mid S \cap \{\mathtt{L}_1,\ldots,\mathtt{L}_h\} \mid + (k - \mid S \cap \{\mathtt{H}_1,\ldots,\mathtt{H}_k\} \mid).$$

C è vero in S se $n \leqslant val(C,S) \leqslant m$.

Un cardinality constraint si usa come un atomo positivo della forma

$$n \ \texttt{\{p(X,Y): range(X,Y) \}} \ m$$

dove `range` è predicato di dominio.
C è vero in S se il numero di atomi della forma $p(X, Y)$ (tali che $range(X, Y)$) è compreso tra n e m.

In luogo delle regole

```
val(4).
val(5).
val(6).
val(7).
```

possiamo scrivere

```
val(4..7).
```

Al posto della regola

```
p:- q(6), q(7), q(8).
```

è possibile la regola

```
p:- q(6..8).
```

(e altre cose usando il ; ...)

- Funzioni built-in: `plus`, `minus`, `times`, `div`, `mod`, `lt`, `gt`, `le`, `ge`, `neq`, `abs`, `and`, `or`,
- Tali funzioni vengono valutate durante il processo di grounding. Quindi in tale momento gli argomenti delle funzioni devono essere disponibili.
- Ci sono sia l'operatore di confronto "==" che quello di assegnamento "=".
- È anche possibile per il programmatore definire proprie funzioni tramite dei programmi C o C++.

ASP SOLVERS

SMODELS

- Supponiamo il programma P stia nel file `asp.lp`.
- Il file viene processato da lparse/smodels invocando il comando

  `lparse asp.lp | smodels` n
- n è un numero intero che indica quanti answer set (al massimo) smodels debba produrre ($n = 0$: tutti)
- Per programmi parametrici, si possono assegnare valori alle costanti (es. lparse -c n=15 asp.lp ...)
- La dichiarazione `#hide p(X,Y).` indica a smodels di non includere gli atomi della forma `p(X,Y)` nell'output prodotto).
  `#hide.` elimina tutti. `#show p(X,Y).` dice di stampare l'output del predicato `p`.

ASP SOLVERS

ELENCO PRINCIPALI SOLVERS

- Smodels (da usare dopo lparse)
- Cmodels (da usare come smodels, dopo lparse)
- Clasp (da usare dopo lparse o meglio dopo il grounder Gringo)
- DLV (grounder integrato)
- GASP: salta il grounding!