Intelligenza Artificiale II

L_{PO} e *logic programming*

Marco Piastra

Clausole di Horn in L_{PO}

- Definizione quasi identica al caso proposizionale

  Forma a clausole (della skolemizzazione di un insieme di formule)

  In ciascuna clausola occorre al massimo un atomo in forma positiva

Fatti, regole e goal

Fatti: clausola con un singolo atomo in forma positiva

$\{Umano(socrate)\}$, $\{Pyramid(x)\}$, $\{Sorella(alba, madreDi(paolo))\}$

Regole: clausola di due o più atomi, uno in forma positiva

$\{Umano(x), \neg Filosofo(x)\}$,
$\forall x\,(Filosofo(x) \rightarrow Umano(x))$

$\{\neg Femmina(x), \neg Genitore(k(x),x), \neg Genitore(k(y),y), Sorella(x,y)\}$
$\forall x \forall y\,((Femmina(x) \wedge \exists z\,(Genitore(z,x) \wedge Genitore(z,y))) \rightarrow Sorella(x,y))$

$\{\neg Above(x,y), On(x,k(x))\}$, $\{\neg Above(x,y), On(j(y),y)\}$
$\forall x \forall y\,(Above(x,y) \rightarrow (\exists z\,On(x,z) \wedge \exists v\,On(v,y)))$

Goal: clausola di atomi in forma negativa

$\{\neg Umano(socrate)\}$

$\{\neg Sorella(alba,x), \neg Sorella(x,paola)\}$

Negazione di $\exists x\,(Sorella(alba,x) \wedge Sorella(x,paola))$

Clausole di Horn e modelli di Herbrand

- Corollario del teorema di Herbrand

  Sia Γ un insieme di clausole di Horn, le seguenti affermazioni sono equivalenti:
  - Γ è soddisfacibile
  - Γ ha un modello di Herbrand

  Non vale in generale: solo se Γ è un insieme clausole di Horn

- **Modello minimo** di Herbrand

  Il modello minimo M_Γ è l'intersezione di tutti i modelli di Herbrand M_i di Γ :

  $M_\Gamma \equiv \cap_{\forall i} M_i$

- Teorema (van Emden e Kowalski, 1976)

  Sia Γ un insieme di clausole di Horn e φ una clausola di Horn,
  le seguenti affermazioni sono equivalenti:
  - $\Gamma \models \varphi$
  - $\varphi \in M_\Gamma$
  - L'unione di tutte le clausole φ che sono conseguenza logica di Γ coincide con M_Γ

Programmi e modello minimo

- Teorema (Apt e van Emden, 1982)

  Sia Π un **programma** (= *un insieme di clausole di Horn*).
  Applicata a Π, la procedura di risoluzione genera il modello minimo M_Π

  La procedura termina se M_Π è finito (raggiungimento del *punto fisso*)

  Esempio:

  $$\Pi \equiv \{\{Umano(x), \neg Filosofo(x)\}, \{Mortale(x), \neg Umano(x)\}, \{Filosofo(socrate)\}, \{Filosofo(platone)\}, \{Filosofo(aristotele)\}\}$$

  Applicando la procedura di risoluzione in modo esaustivo, si ottiene:

  $$\begin{aligned} M_\Pi \equiv \quad & \{\{Mortale(x), \neg Filosofo(x)\}, \\ & \{Filosofo(socrate)\}, \{Filosofo(platone)\}, \{Filosofo(aristotele)\}, \\ & \{Umano(socrate)\}, \{Umano(platone)\}, \{Umano(aristotele)\}, \\ & \{Mortale(socrate)\}, \{Mortale(platone)\}, \{Mortale(aristotele)\}\} \end{aligned}$$

  (assomiglia alla generazione di un database, *implicitamente descritto* da Π...)

Programmi e goal

Un dimostratore di teoremi, applicato ad un programma logico Π, risponde solo a domande del tipo "$\Pi \models \phi$?"

Si rammenti che, se $\Pi \models \phi$, allora $\Pi \cup \{\neg\phi\}$ è insoddisfacibile

- Un sistema di programmazione logica è in grado di generare un particolare sottoinsieme di $\mathbf{M}_\Pi$

Un goal $\{\neg\alpha_1, \neg\alpha_2, ..., \neg\alpha_m\}$, dove occorrono le variabili $x_1, x_2, ..., x_m$

equivale all'enunciato $\forall x_1 \forall x_2 \ldots \forall x_n (\neg\alpha_1 \vee \neg\alpha_2 \vee \ldots \vee \neg\alpha_m)$

che equivale a $\neg\exists x_1 \exists x_2 \ldots \exists x_n (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m)$

Un sistema di programmazione logica genera tutte le **sostituzioni** $[x_1/t_1, x_2/t_2, ..., x_n/t_n]$ tali per cui $\Pi \cup \{\neg(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m)[x_1/t_1, x_2/t_2, ..., x_n/t_n]\}$ è insoddisfacibile

(vale a dire $\Pi \models (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m)[x_1/t_1, x_2/t_2, ..., x_n/t_n]$)

(vale a dire $(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m)[x_1/t_1, x_2/t_2, ..., x_n/t_n] \in \mathbf{M}_\Pi$)

Il goal agisce da filtro, caratterizzando il sottoinsieme di $\mathbf{M}_\Pi$

Goal diverso, sottoinsieme diverso

Esempio

- Un programma logico Π:

$$\Pi \equiv \{\{Umano(x), \neg Filosofo(x)\}, \{Mortale(y), \neg Umano(y)\}, \{Filosofo(socrate)\}, \{Filosofo(platone)\}, \{Filosofo(aristotele)\}\}$$

$$\phi \equiv \exists x\, Mortale(x)$$

$$\neg\phi \equiv \neg\exists x\, Mortale(x) \equiv \forall x\, \neg Mortale(x) \equiv \{\neg Mortale(x)\}$$ (goal in forma di clausola di Horn)

Applicando la procedura di risoluzione in modo esaustivo

Si ottengono le sostituzioni:

$\Sigma \equiv \{[x/socrate], [x/platone], [x/aristotele]\}$

Assomiglia alla query su un database, *implicito* ...

Risoluzione SLD

- Un metodo per la risoluzione di programmi

  S: *selection function*, una funzione di selezione degli atomi da unificare
  L: *linear resolution*, risoluzione lineare, cioè in sequenza
  D: *definite clause*, clausole di Horn con esattamente un letterale positivo

- Descrizione

  Programma (*definite clauses*: regole + fatti): Π

  Regole: $\beta \vee \neg\gamma_1 \vee \neg\gamma_2 \vee \ldots \vee \neg\gamma_n$

  Fatti: δ

  Goal: $\neg\alpha_1 \vee \neg\alpha_2 \vee \ldots \vee \neg\alpha_k$

  Caratteristiche della procedura:

  - I goal vengono considerati secondo l'ordine definito dalla *selection function*
  - Per ciascun goal $\neg\alpha_i$ viene tentata la risoluzione (con unificazione) di tutte le regole (o fatti) che hanno un letterale positivo compatibile (*esplorazione delle alternative*)
  - Le risposte sono le assegnazioni che permettono di derivare la clausola vuota
  - L'insieme delle risposte è un sottoinsieme di M_Π

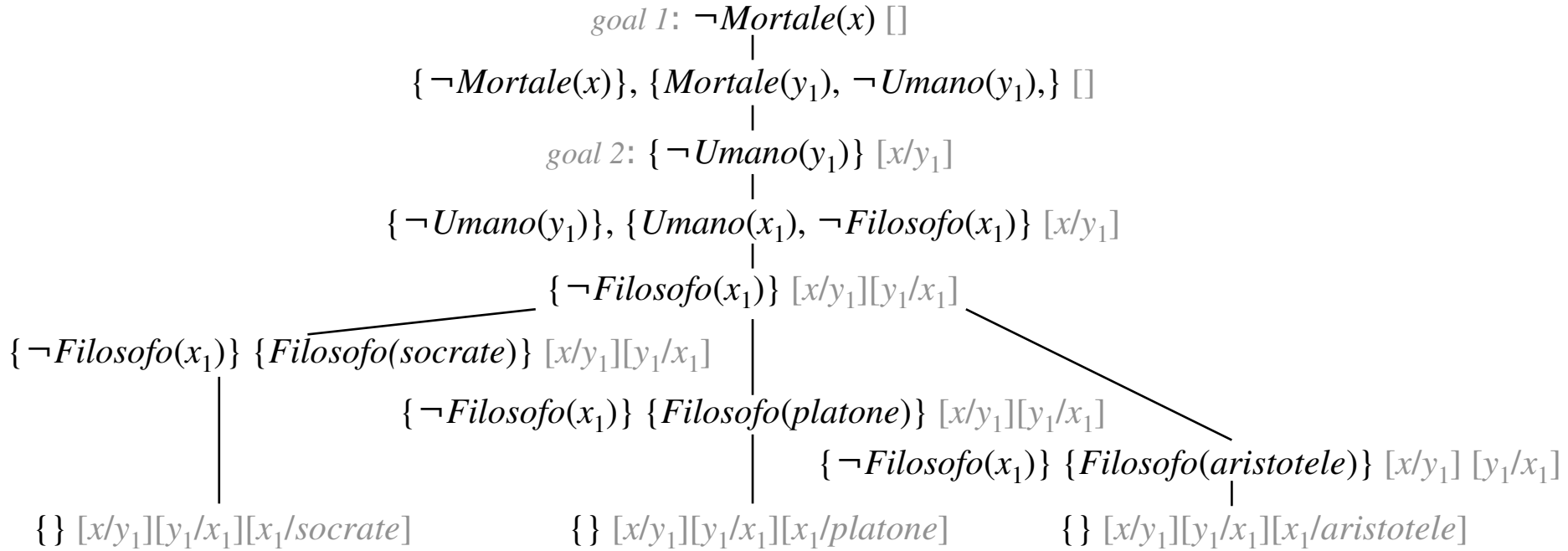
Alberi SLD

- Una traccia del metodo di risoluzione SLD

  Esempio:

  $\Pi \equiv \{\{Umano(x), \neg Filosofo(x)\}, \{Mortale(y), \neg Umano(y)\},$
  $\{Filosofo(socrate)\}, \{Filosofo(platone)\}, \{Filosofo(aristotele)\}\}$

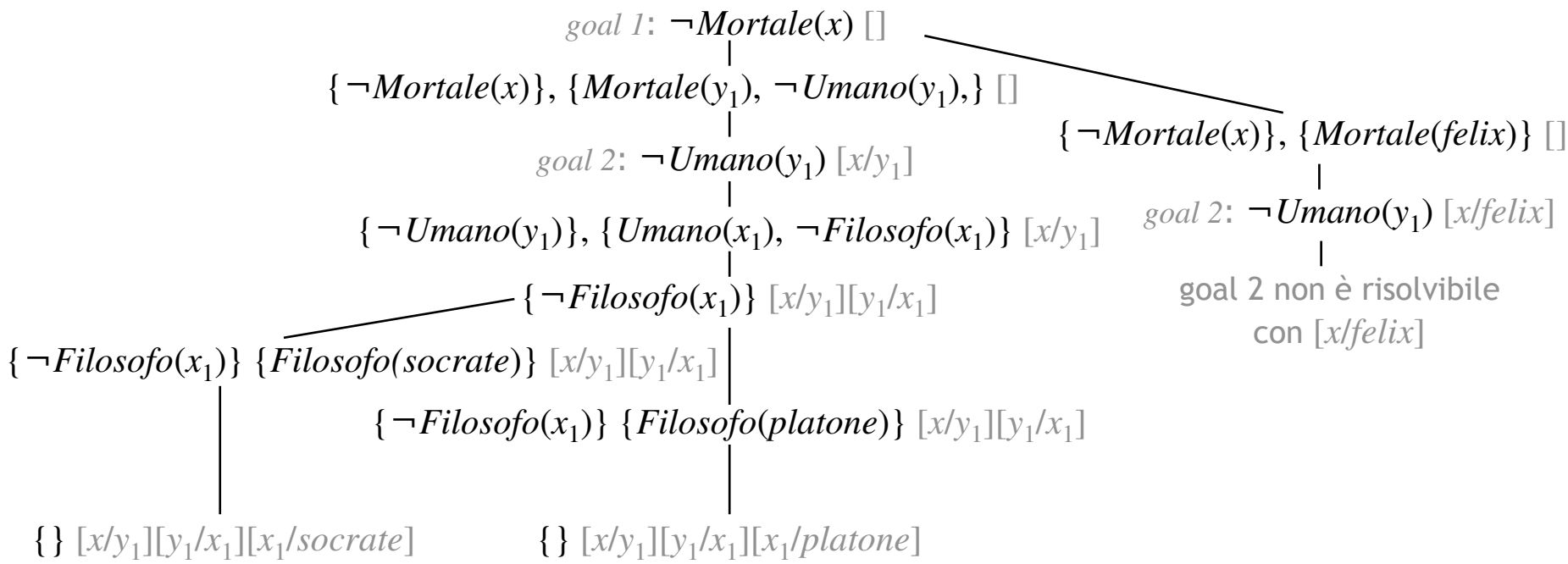
  $goal \equiv \{\neg Mortale(x), \neg Umano(x)\}$ "Chi è mortale ed umano?"



Esempio

- Non tutti i rami SLD si chiudono con successo

$\Pi \equiv \{\{Umano(x), \neg Filosofo(x)\}, \{Mortale(y), \neg Umano(y)\},$
$\{Filosofo(socrate)\}, \{Filosofo(platone)\}, \{Mortale(felix)\}\}$
$goal \equiv \{\neg Mortale(x), \neg Umano(x)\}$ "Chi è mortale ed umano?"



Esempio

- Non tutti gli alberi SLD sono finiti

$\Pi \equiv \{\{Loop(x), \neg Loop(x)\}\}$

$goal \equiv \{\neg Loop(x)\}$

goal: $\neg Loop(x)$ []

|

$\{\neg Loop(x)\}, \{Loop(x_1), \neg Loop(x_1),\}$ []

|

$\{\neg Loop(x_1)\}$ $[x/x_1]$

|

$\{\neg Loop(x_1)\}, \{Loop(x_2), \neg Loop(x_2),\}$ $[x/x_1]$

|

$\{\neg Loop(x_2)\}$ $[x/x_1]$ $[x_1/x_2]$

|

...

SLD e programmazione logica

- **Insieme delle risposte**

  Insieme di tutte le sostituzioni complete delle variabili,
  nei rami dell'albero SLD che si chiudono con successo (= con una clausola vuota)

- Metodo effettivo (*semantica procedurale*)

  S*election function* delle clausole

  Si usa (quasi) sempre la ***leftmost sub-goal first***, con sostituzione del ***sub-goal***

  S*trategia di esplorazione* delle alternative

  - in ***ampiezza*** (*breadth-first*)
  - in ***profondità*** (*depth-first*)

  Il metodo SLD con selezione in *ampiezza* è **completo** (si dice anche SLD **fair**)

  Trova tutti i rami finiti (con successo o meno) dell'albero SLD
  (= *procedura completa di semi-decisione per* $\Pi \models \phi$, con Π e ϕ a clausole)

  In pratica si utilizza la selezione in ***profondità***

  (*Il metodo SLD non è completo - può divergere anche quando* $\Pi \models \phi$)

Risoluzione SLD in Prolog

- Metodo effettivo

  Selection function: leftmost sub-goal first

  Esplorazione *depth-first* delle alternative

  Si esplora una sola alternativa alla volta, e si risparmia memoria (*backtracking*)

  E` una strategia **incompleta**:

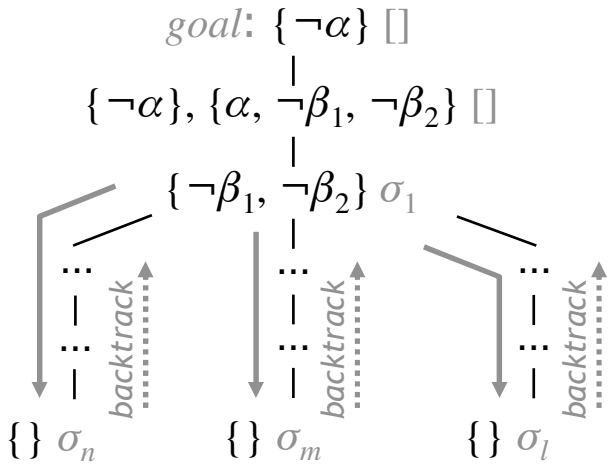
  Un ramo divergente impedisce di trovare tutte le risposte dei rami 'alla destra'

  Scelta tra risoluzioni alternative

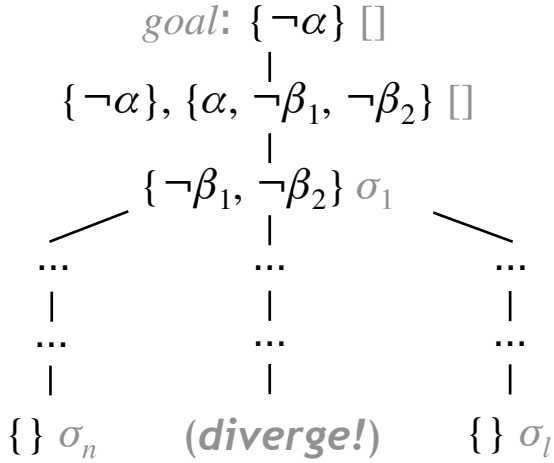
  (= *ordine di esplorazione dei sotto-alberi*)

  Ordine di definizione della clausola applicata

  (≈ quella che compare prima nel file)



Risposte diverse: $\sigma_n\ \sigma_m\ \sigma_l$



Il metodo SLD *depth-first* non troverà la risposta σ_l

Controllo del *backtracking*

- *cut* (!)

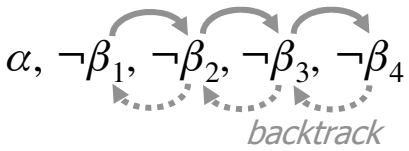
  Interrompe l'esplorazione dell'albero
  al primo successo

  Di fatto, 'taglia' il *backtracking*
  - Prima del cut: backtracking libero
  - Dopo il cut: backtracking libero solo fino al cut

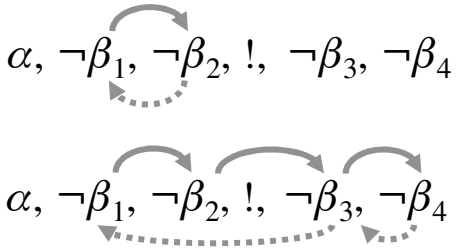
- *fail*

  Forza il fallimento del ramo

cut e *fail* non hanno una semantica logico-dichiarativa: sono un 'controllo di flusso'

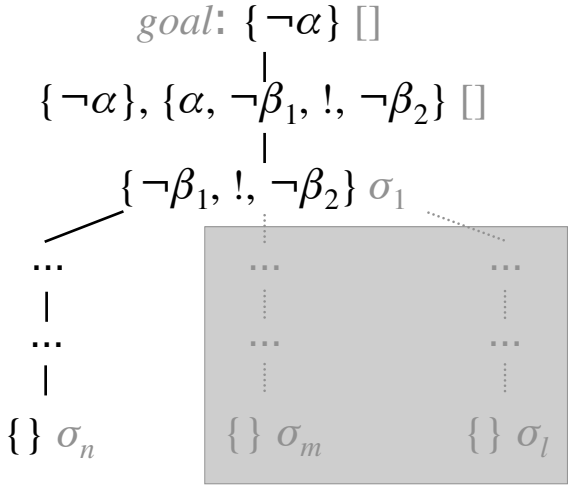


senza *cut*



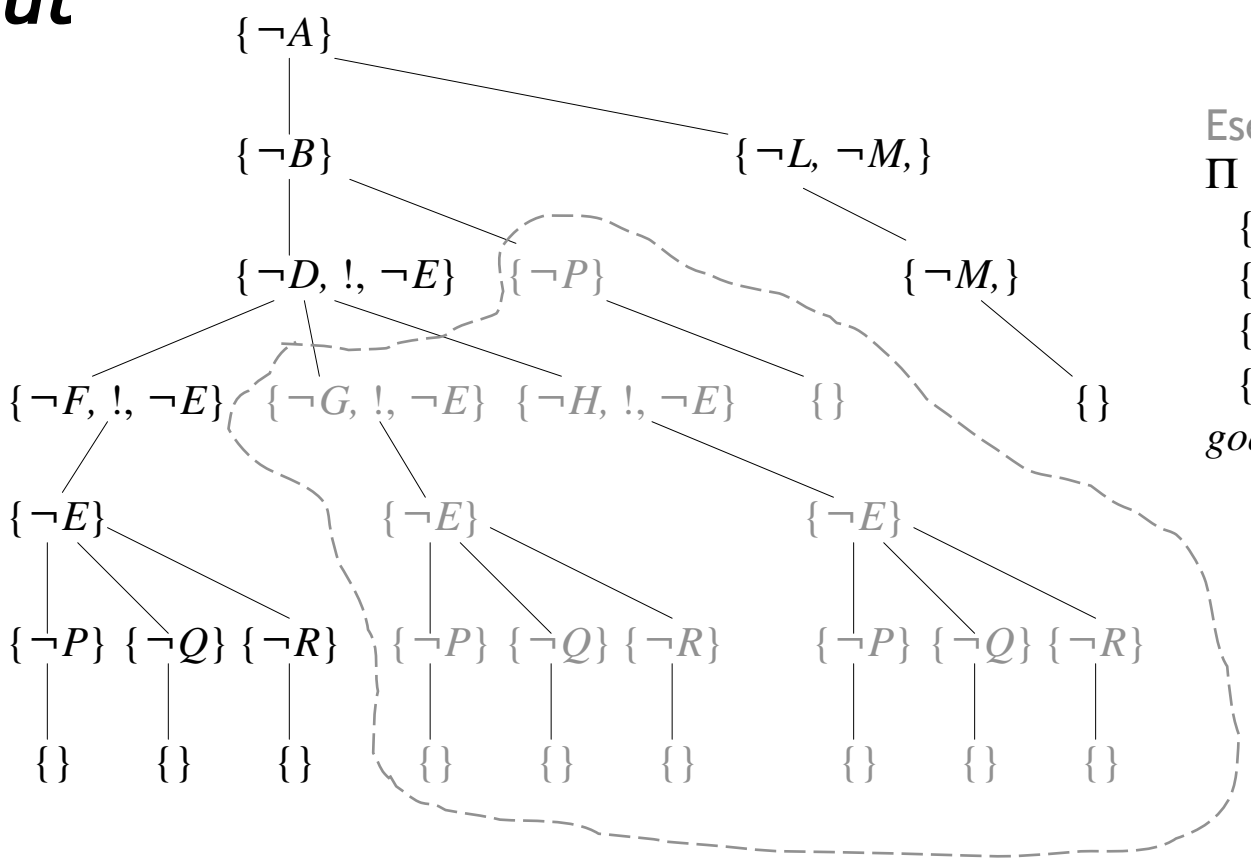
con *cut*

Dal punto di *cut*, il *backtrack* torna alla radice dell'albero SLD (e si arresta).



Per effetto del *cut*, la parte in grigio dell'albero SLD non viene esplorata.

Cut



Questa parte dell'albero SLD
non viene espansa a causa del *cut*

Esempio:

$\Pi \equiv \{\{A, \neg B\}, \{A, \neg L, \neg M,\}$
$\{B, \neg D, !, \neg E\}, \{B, \neg P\},$
$\{D, \neg F\}, \{D, \neg G\}, \{D, \neg H\},$
$\{E, \neg P\}, \{E, \neg Q\}, \{E, \neg R\}, \{L\},$
$\{M\}, \{F\}, \{G\}, \{H\}, \{P\}, \{Q\}, \{R\}\}$
$goal \equiv \{\neg A\}$

Il cut inibisce il *backtracking*
a partire dal goal genitore
(= che attiva la regola che
contiene il *cut*)

Negazione come fallimento

- Clausole in forma negata (\+)

  In generale, nelle clausole di Horn,
  le premesse di una regola
  devono essere in forma positiva

  In Prolog le premesse in forma negativa sono interpretate come negazione per fallimento (*Negation as Failure – NAF, vedi oltre*)

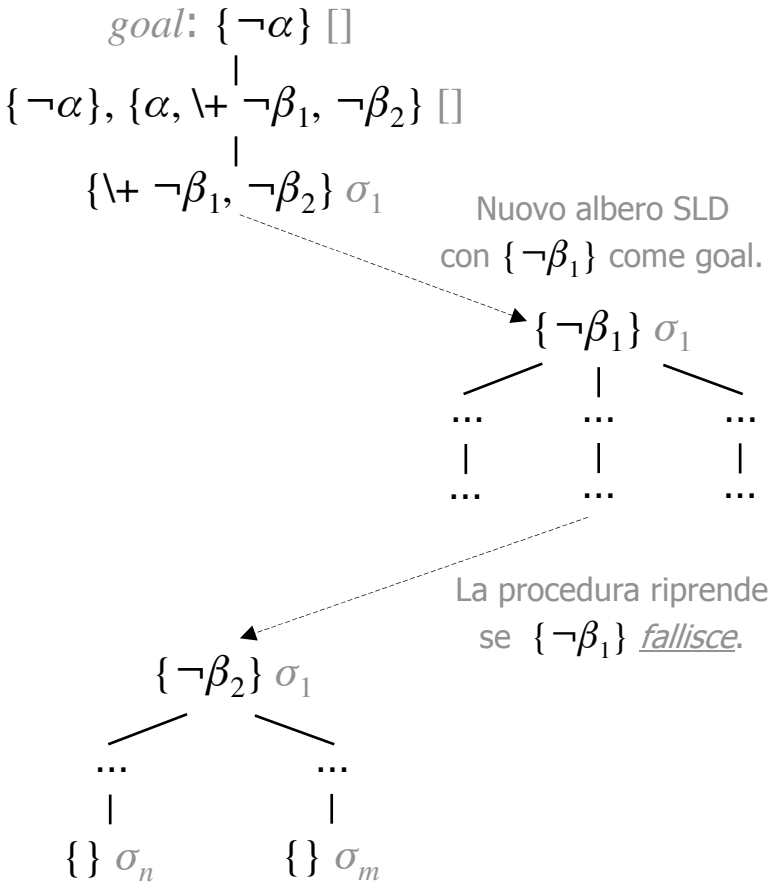
  Per il goal $\backslash+ \neg\beta_1$

  si apre una nuova procedura SLD:
  il goal ha successo se il goal $\neg\beta_1$ fallisce (*senza divergere*)

  Risoluzione SLDNF

  (*Negation as Failure*)

  (Vedere esempio “library.pl”)



Identità

- In Prolog, come viene rappresentata l'identità?
  - Il predicato '=' significa ***unificabilità***
    - t_1 **=** t_2 sse t_1 e t_2 sono unificabili
  - Il predicato '**is**' significa ***unificabilità*** per i valori numerici
    - I valori e funzioni numeriche in Prolog sono trattate in modo speciale:
      - x **is** $(y + 1)$ sse i valori numerici sono identici
- Esempi

```
?- A is 22/7.
A = 3.14286
?- (1 is (2-1)).
Yes
?- (1 = (2-1)).
No
```

- Attenzione:
  - Il predicato '**==**' significa ***equivalenza simbolica***
    - t_1 **==** t_2 sse t_1 e t_2 sono lessicalmente identici

Esempio

- Unificabilità ed identità non sono la stessa cosa

  Esempio (Plaza, 1994)

```
p(X,Y):- \+ X = Y, q(X,Y).
q(a,a).
q(a,b).
?- p(X,Y).
No
p(X,Y):- \+ X == Y, q(X,Y).
q(a,a).
q(a,b).
?- p(X,Y).
Yes [X/a,Y/a]
Yes [X/a,Y/b]
```

X e Y sono sempre unificabili, p.es. [X/Y], quindi il goal negato fallisce

X e Y sono *termini* diversi, quindi il goal negato ha sempre successo

  Attenzione, però, all'ordine dei goal:

```
p(X,Y):- q(X,Y), \+ X = Y.
q(a,a).
q(a,b).
?- p(X,Y).
Yes [X/a,Y/b]
```

Unificazione e *occur check*

- Un altra particolarità del Prolog: omissione dell'*occur check*

  Regola (5) della procedura di costruzione del MGU

(5)	$x = t$ where x does not occur in t and x occurs elsewhere	*apply the substitution* $\{x/t\}$ *to all other equations*

  Il test di occorrenza di x in t è il passo più dispendioso della procedura e viene solitamente disabilitato (o omesso) in Prolog

Risultato:

```
p(X,f(X)).
test :- p(Y,Y).
?- test.
Yes
```

`test` non è derivabile, in quanto non unificabile

(*con un inesistente unificatore* `[X/Y,Y/f(X)]`)

```
q(Y,f(Y)) :- q(Y,Y).
test2 :- q(X,X).
?- test2.
<infinite loop>
```

`test2` non è derivabile, in quanto non unificabile

(*applica la regola* (5) *con* `[Y/f(Y)]`)