

Datalog e negazione



SAPIENZA
UNIVERSITÀ DI ROMA

Lorenzo Pannacci 1948926

Matteo Pannacci 1948942

Clausole di Horn

Una clausola di Horn in un linguaggio di logica del primo ordine è una clausola, ovvero una disgiunzione di literal, dove al più un literal è positivo (atomo o formula atomica) mentre gli altri sono negativi, ovvero negazioni di atomi (preceduti dal simbolo di negazione logica \neg).

Esempio: $p(a) \vee \neg q(a) \vee \neg q(b)$

Consideriamo l'equivalenza logica dell'implicazione materiale ($A \rightarrow B$) con la formula ($\neg A \vee B$) e le leggi di De Morgan: chiamando b_i l' i -esimo literal negativo ed a l'unico literal positivo di una generica clausola di Horn, possiamo riscrivere questa nel seguente modo:

$$a \vee \neg b_1 \vee \neg b_2 \vee \dots \vee \neg b_n \Leftrightarrow a \vee \neg(b_1 \wedge b_2 \wedge \dots \wedge b_n) \Leftrightarrow b_1 \wedge b_2 \wedge \dots \wedge b_n \rightarrow a$$

Esempio: $p(a) \vee \neg q(a) \vee \neg q(b) \Leftrightarrow q(a) \wedge q(b) \rightarrow p(a)$

Chiamiamo l'atomo a 'testa' dell'implicazione e la congiunzione degli atomi b_i 'corpo'.

Clausole di Horn – Fatti e regole ground

Definiamo ‘fatto’ una clausola di Horn composta da un literal positivo e da nessun literal negativo. Questo tipo di clausole è usato per asserire la verità del literal positivo nel mondo che stiamo rappresentando.

Esempio: `uomo(socrate)`

Una clausola composta da un literal positivo ed almeno un literal negativo è invece detta ‘regola’.

Intuitivamente una regola ci permette, attraverso le regole di inferenza di inferire la verità della testa dell’implicazione nel caso in cui il corpo risulti vero.

Una regola è detta ‘regola ground’ se è composta solo da ‘literal ground’, ovvero che non presentano variabili.

Esempio: `uomo(socrate) → mortale(socrate)`

Clausole di Horn – Regole non-ground

Una regola è detta '**regola non-ground**' se possiede literal non-ground e quindi con delle variabili. Le regole non-ground sono da intendersi precedute da quantificatori universali impliciti per ogni variabile con lo scope sull'intera formula e permettono di modellare leggi generali del mondo che stiamo rappresentando.

Esempio: $\text{uomo}(X) \rightarrow \text{mortale}(X) \Leftrightarrow \forall X (\text{uomo}(X) \rightarrow \text{mortale}(X))$

Una regola non-ground r equivale all'insieme $\text{ground}(r)$ di tutte le regole ground ottenibili sostituendo le variabili di r con termini che non presentano variabili. Allo stesso modo dato un insieme di clausole P definiamo l'insieme $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$.

Clausole di Horn – Goal

Una clausola composta da nessun literal positivo ed almeno un literal negativo è detta ‘**clausola di goal**’. E’ tramite queste che linguaggi di programmazione logica che si basano su clausole di Horn possono fare interrogazioni (**query**) sul mondo rappresentato da un insieme di clausole P che chiamiamo programma.

$$\neg b_1 \vee \neg b_2 \vee \dots \vee \neg b_n \Leftrightarrow \neg(b_1 \wedge b_2 \wedge \dots \wedge b_n) \Leftrightarrow b_1 \wedge b_2 \wedge \dots \wedge b_n \rightarrow \text{False}$$

Il funzionamento di una clausola di goal è poco intuitivo. Chiediamo se una congiunzione di literal positivi è logicamente implicata dal programma (i.e. è vera per ogni modello di P) e lo facciamo asserendo il suo negato (una disgiunzione di literal negativi, la nostra «clausola di goal») e vedendo se questo porta ad una contraddizione (i.e. non esiste un modello di P per cui è vera).

Possiamo considerare il caso degenero di una clausola di Horn vuota (senza literal), che risulta sempre falsa:

True \rightarrow False

Clausole di Horn – Goal

Quando compaiono variabili all'interno della congiunzione di literal che vogliamo dimostrare valida la formula è da intendersi preceduta da quantificatori esistenziali, coerentemente con il rapporto che c'è tra negazione e quantificatori dato che la clausola di goal con variabili è implicitamente preceduta da quantificatori universali.

Nel caso in cui gli literal utilizzino come termini delle variabili, invece di ritornare se la formula sia valida può essere di interesse conoscere per quali valori delle variabili specificate dalla formula questa risulta vera.

Esempio:

uomo(socrate)

uomo(platone)

uomo(X) \rightarrow mortale(X)

Query: mortale(socrate)

Risultato: True

Query: $(\exists X)$ mortale(X)

Risultato: True (X = socrate, platone)

Positive Logic Programs (PLP)

Definiamo 'Positive Logic Program' (o Horn Logic Program) un insieme finito di fatti e regole sotto forma di clausole di Horn in un linguaggio di logica del primo ordine. Nel contesto dei programmi logici usiamo una sintassi diversa che espliciti il significato di «per mostrare a mostro b_1, b_2, \dots, b_n »:

$$a \leftarrow b_1, b_2, \dots, b_n \quad (n \geq 0)$$

Dove a, b_1, b_2, \dots, b_n sono atomi in un linguaggio di logica del primo ordine. Notiamo che l'atomo a non è opzionale perciò un PLP non può includere clausole di tipo goal.

Esempio: il programma considerato nell'esempio precedente è un PLP, ma adottiamo la nuova sintassi:

uomo(socrate)

uomo(platone)

mortale(X) \leftarrow uomo(X)

Per attribuire un significato ad un programma logico utilizziamo delle **semantiche**, definite attraverso il concetto di **modello canonico**. Il percorso è quasi obbligato per classi di problemi più semplici come PLP, ma generalizzando troveremo diverse semantiche con proprietà differenti.

Universo, Base ed Interpretazione di Herbrand

Per trattare di semantiche e modelli di un programma logico formalizziamo alcuni concetti. Dato un certo programma logico P definiamo:

- **Universo di Herbrand (HU):** l'insieme di tutti i termini ground (i.e. che non contengono variabili) esprimibili in P , ovvero tutte le costanti in P e tutti simboli di funzione in P aventi come operandi a loro volta termini ground. Notiamo che questo insieme è infinito sse P contiene almeno un simbolo di funzione di arità > 0 . In questa presentazione tralasceremo i simboli di funzione per semplicità e per rimanere più coerenti con le capacità di Datalog.
- **Base di Herbrand (HB):** l'insieme di tutti gli atomi ground, ovvero l'insieme di tutti i predicati aventi come operandi elementi appartenenti all' $HU(P)$.
- **Interpretazione di Herbrand (I):** un sottoinsieme di $HB(P)$ che contiene tutti e soli gli atomi ground veri in una certa situazione del 'mondo' preso in considerazione.

Herbrand – Esempio

Consideriamo il seguente programma P_1 (PLP):

$p(a, b)$

$q(b)$

$r(X, Y) \leftarrow p(X, Y), q(Y)$

Da cui otteniamo:

- $HU = \{a, b\}$
- $HB = \{p(a, a), p(a, b), p(b, a), p(b, b), q(a), q(b), r(a, a), r(a, b), r(b, a), r(b, b)\}$

Consideriamo alcune interpretazioni, ad esempio:

- $I_1 = \emptyset$
- $I_2 = \{p(a, b), q(b)\}$
- $I_3 = \{p(a, b), q(b), r(a, b)\}$
- $I_4 = \{p(a, b), q(b), r(a, b), p(b, a)\}$

Herbrand – Interpretazioni e modelli

Analizziamo le interpretazioni considerate nell'esempio precedente:

$I_1 = \emptyset$: questa interpretazione è evidentemente in contraddizione con P_1 in quanto non contiene i due fatti esplicitati dal programma stesso, ovvero $p(a, b)$ e $q(b)$.

$I_2 = \{p(a, b), q(b)\}$: questa interpretazione contiene i fatti esplicitati da P_1 , ma non ciò che questi implicano tramite la terza clausola, $r(a, b) \leftarrow p(a, b), q(b)$. Anche questa interpretazione è contraddizione con il programma.

$I_3 = \{p(a, b), q(b), r(a, b)\}$: questa interpretazione è corretta, è compatibile con P_1 perché non è in contraddizione con nessuna clausola del programma. Chiamiamo interpretazioni di questo tipo **modelli**.

$I_4 = \{p(a, b), q(b), r(a, b), p(b, a)\}$: notiamo che anche questa interpretazione non è in contraddizione con nessuna clausola di P_1 , perciò è un altro modello per il programma.

Atomi infondati ed Insiemi infondati

Cosa cambia tra i due modelli I_3 e I_4 che li rende differenti ma entrambi compatibili con il programma? La differenza è nella presenza dell'atomo $p(b, a)$: questo non in contraddizione con P_1 , tuttavia non segue dagli altri atomi presenti in I_4 , diciamo perciò che $p(b, a)$ è un '**atomo infondato**' per I_4 .

Intuitivamente un atomo è infondato per l'interpretazione I di un programma P quando non è un fatto e per ogni regola da cui può essere derivato o la premessa non è rispettata o questa contiene un atomo infondato.

Formalizziamo questo concetto: dato un Positive Logic Program P ed una sua interpretazione I , l'insieme $U \subseteq HB(P)$ è detto '**insieme infondato**' di P relativamente ad I se per ogni atomo $a \in U$, per ogni clausola ground $r \in \text{ground}(P)$ avente come head a esiste un atomo b nel body di r tale che $b \notin I$ oppure $b \in U$.

Dati P ed I indichiamo con $U_P(I)$ il più grande insieme infondato di P relativamente ad I .

Invertendo la definizione diciamo che un atomo a è '**fondato**' per una interpretazione I di un programma P quando è un fatto oppure esiste una regola ground con a come head e nel body solo atomi fondati per I .

Positive Logic Programs – Minimal model semantics

Osserviamo che se I è un modello di P allora l'interpretazione $I' = I - U_P(I)$ è a sua volta modello di P e questo modello contiene tutti e solo gli atomi fondati per I .

Per come lo abbiamo definito nei Positive Logic Programs il concetto fondatezza di un atomo (rispetto ad un modello) è equivalente alla **necessarietà dell'atomo** per il programma: un'interpretazione non può essere un modello se non contiene quell'atomo.

Definiamo la Minimal Model Semantics, che sceglie come modello canonico quello che minimizza il numero di atomi presenti nel modello. Chiamiamo '**modello minimo**' per un programma P un modello di P tale che nessun suo sottoinsieme proprio sia a sua volta modello di P .

Per quanto detto prima affinché un'interpretazione sia un modello deve contenere (almeno) tutti gli atomi fondati (per qualunque modello) inoltre come abbiamo visto se rimuovo gli atomi infondati quello che ottengo è sempre un modello, perciò il modello minimo contiene tutti e solo gli atomi fondati per qualunque modello.

Esempio: per il programma P_1 l'interpretazione $I_3 = \{p(a, b), q(b), r(a, b)\}$ è un modello minimo.

Positive Logic Programs – Proprietà del modello minimo

Osserviamo alcune proprietà riguardo i modelli dei PLP:

- Dati due modelli I e J di un programma P anche $I \cap J$ è un modello di P .
- L'intersezione di tutti i modelli di un programma P è il modello minimo di P , ovvero il modello minimo è contenuto in ogni modello.

Altra proprietà interessante è che l'interpretazione $I = HB(P)$ è sempre un modello di P . Ogni fatto è sicuramente soddisfatto in quanto l'interpretazione contiene ogni atomo possibile e per ogni regola il corpo è rispettato ma come per i fatti lo è sicuramente anche la testa.

Questa interpretazione può essere usata come modello banale per ricavare il modello minimo: ci basta calcolare l'insieme infondato per questo modello e sottrarlo al modello stesso.

Ambiguità nella definizione di fondatezza

Esempio: consideriamo il seguente programma ed una sua interpretazione

person(a)

person(b)

guilty(a) \leftarrow person(a), guilty(b)

guilty(b) \leftarrow person(b), guilty(a)

$I = \{\text{person(a)}, \text{person(b)}, \text{guilty(a)}, \text{guilty(b)}\}$

Andando a studiare la fondatezza rispetto ad I dell'atomo *guilty(a)* troviamo che questo è fondato sse l'atomo *guilty(b)* è fondato. Analogamente *guilty(b)* è fondato sse lo è *guilty(a)*. Perciò *guilty(a)* è fondato sse *guilty(a)* è fondato.

L'ambiguità descritta è dovuta alla presenza di definizioni «autosussistenti» che sono paradossali, non desiderabili e non aggiungono espressività al programma quindi andrebbero evitate. Osserviamo che I è un modello ma non è minimo, se togliamo sia *guilty(a)* sia *guilty(b)* quello che otteniamo è ancora un modello.

Algoritmo di computazione del modello minimo

Mostriamo un algoritmo costruttivo per calcolare il modello minimo di un Positive Logic Program P .

Definiamo l'operatore di 'conseguenza immediata' T_P che a partire da un'interpretazione I di P genera una nuova interpretazione I' contenente tutti gli atomi in I e gli atomi da questi implicati secondo le regole di P .

L'algoritmo è una applicazione ripetuta di T_P che inizialmente prende come interpretazione i fatti in P ed ad ogni iterazione successiva l'interpretazione precedentemente computata. L'algoritmo termina quando l'interpretazione di input e di output per una iterazione sono uguali (i.e. abbiamo raggiunto un 'punto fisso').

Esempio: consideriamo il programma P_2 :

$p(a) \leftarrow p(b)$

$p(b) \leftarrow p(c), p(d)$

$p(c) \leftarrow p(e)$

$p(d)$

$p(e)$

Applichiamo l'algoritmo:

$I_0 = \{p(d), p(e)\}$

$I_1 = T_P(I_0) = \{p(d), p(e), p(c)\}$

$I_2 = T_P(I_1) = \{p(d), p(e), p(c), p(b)\}$

$I_3 = T_P(I_2) = \{p(d), p(e), p(c), p(b), p(a)\}$

$I_4 = T_P(I_3) = \{p(d), p(e), p(c), p(b), p(a)\}$ (modello minimo)

Prolog e Datalog

La **programmazione dichiarativa** è un paradigma di programmazione, in linguaggi di questo tipo si specifica **cosa** deve essere fatto piuttosto che **come** deve essere fatto. Questo è in contrasto con i classici linguaggi di programmazione imperativi che invece si occupano di specificare il flusso di controllo del programma.

La **programmazione logica** è un altro paradigma di programmazione, considerato una specializzazione di quella dichiarativa. Si basa sulla logica: un problema viene definito in termini di fatti e regole ed attraverso l'uso di un motore di inferenza vengono dedotte conclusioni sul mondo trattato.

Prolog (dal francese '*PRO*grammation en *LOG*ique') è un linguaggio di programmazione logico implementato per la prima volta nel 1972 ed è ad oggi il più popolare. Inizialmente l'idea dietro Prolog era quella di un «dimostratore di teoremi» capace di lavorare su clausole di Horn ma nel tempo è stato espanso fino a diventare un linguaggio Turing-completo.

Datalog è sintatticamente un sottoinsieme di Prolog, perde simboli di funzione, l'operatore 'cut' e la negazione (concetto che approfondiremo), quello che rimane è la possibilità di definire un insieme finito di fatti e regole in forma di clausole di Horn. E' utilizzato come linguaggio di query per database deduttivi.

Prolog e Datalog

Mentre sia Prolog che Datalog seguono chiaramente il paradigma della programmazione logica il discorso per quella dichiarativa è più complicato. Nonostante Prolog abbia molte proprietà dei linguaggi dichiarativi e venga comunemente definito come tale il fatto che l'ordine in cui sono scritte le clausole possa modificare il risultato di un programma (basti pensare all'operatore 'cut') lo avvicina ai linguaggi imperativi.

Per Datalog questo non è vero: l'ordine in cui sono espresse le istruzioni è ininfluenza sul risultato; per questo possiamo definirlo come un linguaggio **'puramente dichiarativo'**.

La sintassi di una clausola di Horn in Datalog/Prolog riprende la sintassi definita per i PLP, ciò che cambia è solamente che l'implicazione « \leftarrow » diventa « $:-$ » ed ogni clausola termina con un punto « $.$ » :

$$a \leftarrow b_1, b_2, \dots, b_n$$

$$a \text{ :- } b_1, b_2, \dots, b_n.$$

Datalog positivo

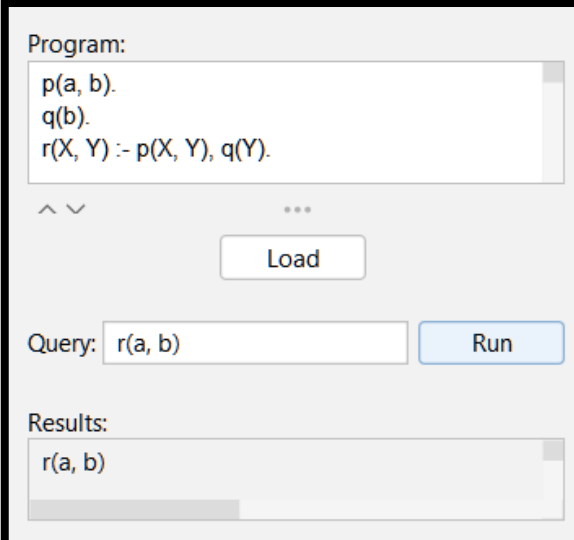
Datalog positivo è il termine con cui ci riferiamo al Datalog senza estensioni che corrisponde ai Positive Logic Programs.

Mostriamo come esempio di Datalog Positivo **ABCDatalog**, una implementazione di Datalog open-source scritta in Java che comprende sia un motore di inferenza bottom-up che top-down.

L'**approccio bottom-up** è una versione raffinata dell'algoritmo descritto per il calcolo del modello minimo: dai fatti deriva tutte le conseguenze, crea il modello e valuta la query.

L'**approccio top-down** parte dalla query e cerca di dimostrarne tutti i punti.

I due approcci hanno i loro punti di forza e debolezze ma solitamente per Datalog positivo viene preferito l'approccio bottom-up.



Program:

```
p(a, b).  
q(b).  
r(X, Y) :- p(X, Y), q(Y).
```

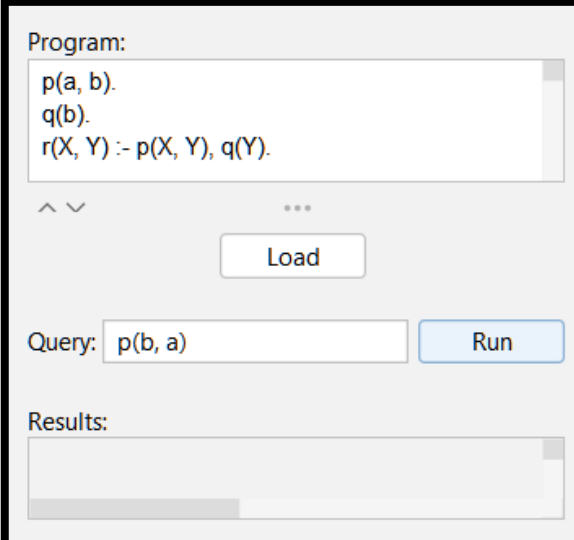
^ v ...

Load

Query: r(a, b) Run

Results:

```
r(a, b)
```



Program:

```
p(a, b).  
q(b).  
r(X, Y) :- p(X, Y), q(Y).
```

^ v ...

Load

Query: p(b, a) Run

Results:

Datalog positivo – Rapporto con l'algebra relazionale

Come abbiamo detto Datalog positivo non è Turing-completo. In particolare ne possiamo confrontare l'espressività con un altro linguaggio di query come l'algebra relazionale (o analogamente il calcolo relazione). Sono dimostrabili i seguenti teoremi:

- L'algebra relazionale non è più espressiva di Datalog.
- Datalog non è più espressivo dell'algebra relazionale.
- Datalog è più espressivo dell'algebra relazionale senza l'operatore di differenza.

Questo significa che l'espressività di Datalog e dell'algebra relazionale non sono direttamente confrontabili. Intuitivamente ciò è giustificabile dal fatto che Datalog permette la ricorsione che non è possibile nell'algebra relazionale mentre l'algebra relazionale permette la complementarità (tramite l'operatore differenza) che non è possibile in Datalog.

Perciò per avere una maggiore espressività possiamo estendere Datalog con il concetto di complementarità, che possiamo rendere tramite la negazione.

Concetto di Negazione

Tramite i Positive Logic Programs siamo in grado a modellare diversi tipi di problemi, tuttavia non riusciamo ad esplicitare concetti facilmente esprimibili in linguaggio naturale usando il significato intuitivo di negazione.

Esempio: consideriamo l'affermazione «Una persona che non ha commesso il crimine è innocente»; questa può essere espressa in logica del prim'ordine come « $\forall X (\text{person}(X) \wedge \neg \text{crime}(X) \rightarrow \text{innocent}(X))$ », tuttavia in virtù dei limiti che abbiamo restringendoci alle sole clausole di Horn non riusciamo ad esprimere questo concetto in un Positive Logic Program.

Da qui arriva la volontà di **espandere l'espressività del linguaggio** aggiungendo il concetto di negazione, cosa che esploreremo facendo particolare attenzione alle conseguenze che l'allontanarsi da un modello semplice come i Positive Logic Programs comporta.

Nel contesto della programmazione logica il **concetto di negazione è ambiguo**, distinguiamo due tipologie distinte di negazione note come «negazione forte» e «negazione debole».

Negazione Forte e Negazione Debole

La **negazione forte** corrisponde all'idea di negazione classica (o negazione logica) ed è indicata tramite il simbolo « \neg » preposto all'atomo da negare. L'atomo negato (fortemente) « $\neg p(a)$ » risulta vero quando è possibile dimostrare che l'atomo positivo « $p(a)$ » sia falso.

La **negazione debole**, anche nota come 'Negation as failure' o 'Default negation', è un concetto adottato principalmente nel contesto della programmazione logica. E' indicata tramite il simbolo «**not**» preposto all'atomo da negare.

L'atomo negato (debolmente) «*not* $p(a)$ » risulta vero se non è possibile dimostrare la verità dell'atomo positivo « $p(a)$ ». Ciò significa che un atomo è considerato falso «di default», fino a prova contraria.

Esempio:

$\text{innocent}(a) \leftarrow \text{not crime}(a)$

$\text{innocent}(a) \leftarrow \neg \text{crime}(a)$

La prima afferma che « a è innocente se non è dimostrabile che abbia commesso il crimine» mentre la seconda afferma che « a innocente se è dimostrabile che non ha commesso il crimine».

Normal Logic Programs (NLP)

Tramite il concetto di negazione debole possiamo generalizzare i Positive Logic Programs. Definiamo un Normal Logic Program (o General Logic Program) come un insieme di formule del tipo:

$$a \leftarrow b_1, b_2, \dots, b_n, \text{not } c_1, \text{not } c_2, \dots, \text{not } c_m \quad (n, m \geq 0)$$

Dove $a, b_1, b_2, \dots, b_n, c_1, c_2, \dots, c_m$ sono atomi in un linguaggio di logica del primo ordine. Chiamiamo b_1, b_2, \dots, b_n **literal positivi** e $\text{not } c_1, \text{not } c_2, \dots, \text{not } c_m$ **literal negativi**. Stiamo permettendo l'uso della negazione (debole) unicamente per gli atomi nei body delle clausole. Osserviamo che queste non sono più in generale clausole di Horn, perciò le proprietà dei PLP derivanti da questa restrizione non valgono.

Esempio: P_2

person(a)

crime(a)

guilty(X) \leftarrow person(X), crime(X)

innocent(X) \leftarrow person(X), *not* guilty(X)

Query: guilty(a) > True

Query: innocent(a) > False

Normal Logic Programs – Esempi

Esempio: P_3

person(a)

guilty(X) \leftarrow person(X), *not* innocent(X)

innocent(X) \leftarrow person(X), *not* guilty(X)

$M_{p3,1} = \{\text{person(a), innocent(a)}\}$

$M_{p3,2} = \{\text{person(a), guilty(a)}\}$

Query: innocent(a) > ?

Query: guilty(a) > ?

Esempio: P_4

person(a)

guilty(X) \leftarrow person(X), crime(X)

innocent(X) \leftarrow person(X), *not* guilty(X)

$M_{p4,1} = \{\text{person(a), innocent(a)}\}$

$M_{p4,2} = \{\text{person(a), guilty(a), crime(a)}\}$

Query: innocent(a) > ?

Query: guilty(a) > ?

Non riusciamo a rispondere a queste query perché esistono diversi modelli dei programmi che soddisfano la proprietà di modello minimo, non trovandoci più nella classe dei Positive Logic Programs.

Per dare un significato al programma dobbiamo adottare una nuova semantica.

Normal Logic Programs – Osservazione

Come per i Positive Logic Programs l'interpretazione $I = HB(P)$ è un modello per il Normal Logic Program P .

Ogni fatto è soddisfatto in quanto l'interpretazione contiene ogni possibile atomo, ogni regola con almeno un atomo negato nel body è falsificata e quindi non in contraddizione con l'interpretazione ed ogni regola con solo atomi positivi nel body è soddisfatta perché la conseguenza è sicuramente nell'interpretazione.

Da questo ne consegue che ogni Normal Logic Program ha almeno un modello.

Il grande scisma della programmazione logica

L'esistenza di molteplici modelli minimi per uno stesso programma ha portato alla nascita di diversi approcci per definire la semantica di un Normal Logic Program. Questa divisione prende il nome di «Grande Scisma della Programmazione Logica».

Dividiamo questi approcci in due famiglie:

- **Single Models Semantics:** la prima idea è mantenere l'unicità del modello canonico ridefinendolo. Sono state proposte nella letteratura due tipologie di modello canonico:
 - Modello perfetto (stratificazione)
 - Well-founded model
- **Multiple Model Semantics:** l'alternativa è abbandonare l'idea di modello unico ed invece permettere l'esistenza di molteplici modelli canonici.
 - Modello stabile

Datalog semi-positivo

In modo analogo a quanto fatto per i Logic Programs, possiamo estendere i programmi in Datalog con l'utilizzo della negazione (debole) nel body. I programmi in questo 'Datalog con Negazione' sono equivalenti ai Normal Logic Programs.

Datalog semi-positivo è un approccio particolarmente semplificato alla negazione in Datalog. Nel contesto dell'utilizzo di Datalog come linguaggio di query su database dividiamo i predicati in due classi disgiunte:

- **EDB** (extensional database): la relazione è contenuta nel database che contiene i fatti
- **IDB** (intensional database): la relazione è definita da una o più regole

Datalog semi-positivo permette l'utilizzo della negazione debole ma unicamente applicata su predicati EDB. Prima dell'esecuzione viene calcolato un nuovo predicato come il complementare del predicato EDB che è stato negato (di fatto esprimendo una negation as failure) e viene utilizzato al suo posto.

Una volta effettuate queste sostituzioni il programma può essere eseguito come un qualsiasi programma definito su Datalog positivo.

Stratified Negation – Grafo delle dipendenze

Un grande problema che si ha nella computazione di un modello in un programma con negazione debole è che in generale non è possibile essere sicuri della non dimostrabilità di un atomo in un punto intermedio della computazione e ciò impedisce l'applicazione di regole aventi nel body la negazione dell'atomo.

Per poter applicare la regola $\llbracket a \leftarrow b_1, b_2, \dots, b_n, \text{not } c_1, \text{not } c_2, \dots, \text{not } c_m \rrbracket$ ed in caso positivo inferire l'atomo a , i valori degli atomi $b_1, b_2, \dots, b_n, c_1, c_2, \dots, c_m$ devono essere tutti noti.

Perciò una regola di un programma definisce una dipendenza tra il predicato della testa ed i predicati del body. Queste dipendenze possono essere modellizzate tramite un '**grafo delle dipendenze**'.

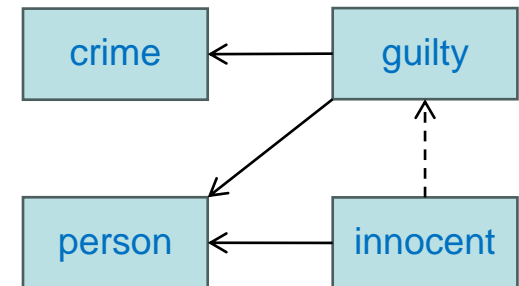
Esempio: riprendiamo il programma P_4

$\text{person}(a)$

$\text{guilty}(X) \leftarrow \text{person}(X), \text{crime}(X)$

$\text{innocent}(X) \leftarrow \text{person}(X), \text{not guilty}(X)$

(la freccia tratteggiata indica la dipendenza dalla negazione dell'atomo)



Stratified Negation – Stratificazione

Possiamo definire la stratificazione di un programma P come una partizione $\Sigma = \{S_1, S_2, \dots, S_n\}$ dei predicati in esso presenti tale da rispettare le seguenti proprietà:

- Se $p \in S_i$, $q \in S_j$ e p dipende da q non negato allora $i \geq j$
- Se $p \in S_i$, $q \in S_j$ e p dipende da q negato allora $i > j$

In questo modo **la stratificazione specifica un ordine di valutazione dei predicati tale da risolvere il problema delle dipendenze**, ovvero in modo che predicati dipendenti siano svolti dopo tutti i predicati da cui dipendono, stiamo obbligando il programma logico a seguire un certo ordine di esecuzione.

Osserviamo che la stratificazione di un programma non è unica.

Esempio: stratificazione di P_4

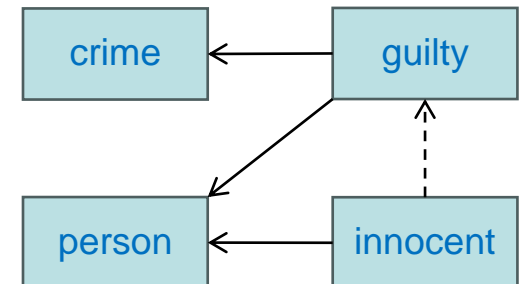
$S_1 = \{\text{person, crime}\}$

$S_2 = \{\text{guilty}\}$

$S_3 = \{\text{innocent}\}$

$S'_1 = \{\text{person, crime, guilty}\}$

$S'_2 = \{\text{innocent}\}$



Algoritmo di computazione del modello perfetto

Mostriamo un algoritmo che generalizza il calcolo del modello minimo iterandone il ragionamento per ogni strato definito sul programma.

Dato un Normal Logic Program P e la sua stratificazione $\Sigma = \{S_1, S_2, \dots, S_n\}$ definiamo P_{S_i} , $i \in \{1, \dots, n\}$ come i sottoinsiemi di P che presentano tutte le regole aventi come testa i predicati appartenenti allo strato i .

L'algoritmo si limita ad eseguire in sequenza la computazione del modello minimo per $P_{S_1}, P_{S_2}, \dots, P_{S_n}$ usando come interpretazione iniziale l'output dell'iterazione precedente e per P_{S_1} l'insieme di tutti fatti.

I literal negativi vengono calcolati grazie alla regola di inferenza della negazione debole a partire dall'interpretazione corrente (se l'atomo non appartiene all'interpretazione corrente il suo literal negato è considerato vero) ed il corretto funzionamento è garantito dalla stratificazione del programma.

Il risultato dell'algoritmo è un modello minimo per P , anche se non necessariamente l'unico. Chiamiamo questo modello **'modello perfetto'**.

Stratified Negation – Esempio

Riprendiamo il programma P_4 con la stratificazione $\Sigma = \{ S_1 = \{\text{person, crime}\}, S_2 = \{\text{guilty}\}, S_3 = \{\text{innocent}\} \}$.

Calcoliamo i sottoinsiemi di regole relativi agli strati:

$$P_{s1} = \{ \}$$

$$P_{s2} = \{ \text{guilty}(X) \leftarrow \text{person}(X), \text{crime}(X) \}$$

$$P_{s3} = \{ \text{innocent}(X) \leftarrow \text{person}(X), \text{not guilty}(X) \}$$

Eseguiamo l'algoritmo:

$$M_1 = \{ \text{person}(a) \} \quad (\text{contiene solo i fatti, in quanto } P_{s1} \text{ è vuoto})$$

$$M_2 = \{ \text{person}(a) \} \quad (\text{l'iterazione non aggiunge nuovi atomi})$$

$$M_3 = \{ \text{person}(a), \text{innocent}(a) \} \quad (\text{possiamo affermare la falsità di } \text{guilty}(a) \text{ perché non presente in } M_2)$$

Il modello perfetto del programma è $M = \{ \text{person}(a), \text{innocent}(a) \}$

Qualsiasi stratificazione porta allo stesso modello perfetto, questo è unico per un certo programma P .

Notiamo che esiste un altro modello minimo $M' = \{ \text{person}(a), \text{guilty}(a), \text{crime}(a) \}$.

Estensione del concetto di insieme infondato

Sia M che M' sono modelli minimi per il programma, perché scegliamo il primo come modello perfetto?

Sia $innocent(a)$ sia la coppia « $crime(a)$, $guilty(a)$ » **non sono necessariamente veri** in quanto esiste almeno un modello che non li contiene, riprendendo però la definizione di infondatezza osserviamo che la seconda alternativa risulta infondata, in quanto $crime(a)$ è infondato e $guilty(a)$ è inferito da questo.

Per studiare la fondatezza del primo modello dobbiamo invece estendere la definizione di insieme infondato: Dato un Normal Logic Program P ed una sua interpretazione I , l'insieme $U \subseteq HB(P)$ è detto **insieme infondato** di P relativamente ad I se per ogni atomo $a \in U$, per ogni clausola ground $r \in ground(P)$ avente come testa a è vera (almeno) una delle due affermazioni:

- Esiste un literal positivo b che appare nel body di r tale che $b \notin I$ oppure $b \in U$.
- Esiste un literal negativo $not\ c$ nel body di r tale che $c \in I$.

Dalla nuova definizione segue che il primo è un **modello fondato**, nel senso che è un modello il cui insieme infondato è vuoto. Per problemi stratificabili il modello fondato è unico e coincide con il modello perfetto.

Stratified Negation - Monotonicità

L'introduzione della negazione rende la logica trattata non più **monotona**. Mentre per i Positive Logic Programs l'introduzione di nuove clausole non poteva cambiare il risultato di una interrogazione da vero a falso ciò non vale più per i Normal Logic Programs. Parliamo perciò di «*non-monotonic reasoning*».

Esempio:

Riprendiamo il programma P_4 .

person(a)

guilty(X) \leftarrow person(X), *not* innocent(X)

innocent(X) \leftarrow person(X), *not* guilty(X)

Usando la semantica definita la query «innocent(*a*)» risulterà vera. Però se aggiungiamo al programma la clausola «*crime(a)*» la stessa query diventerà falsa.

Stratified Negation – Limite del modello perfetto

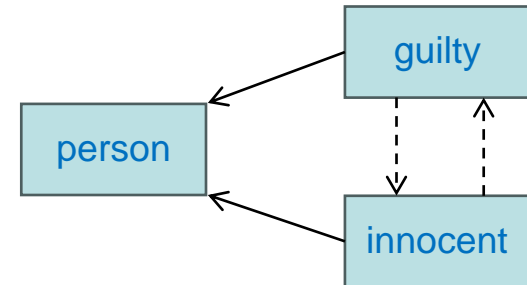
Non sempre è possibile trovare una stratificazione per un Normal Logic Program. Se un programma presenta cicli di dipendenza in cui è presente almeno una negazione esistono dei predicati che devono trovarsi allo stesso tempo sia prima che dopo altri predicati, rendendo impossibile trovare un ordine corretto.

Esempio: riprendiamo il programma P_3

`person(a)`

`guilty(X) ← person(X), not innocent(X)`

`innocent(X) ← person(X), not guilty(X)`



L'utilizzo della negazione stratificata ci obbliga quindi a **restringere la classe dei programmi** su cui possiamo lavorare ai soli Normal Logic Programs stratificabili.

Datalog stratificato

La semantica del modello perfetto può essere applicata a Datalog esteso con la negazione, ciò che otteniamo è detto **Datalog stratificato** e può essere visto come una generalizzazione di Datalog semi-positivo.

Questa semantica è quella più utilizzata nelle moderne implementazioni di Datalog: **Soufflé** (considerato lo stato dell'arte), **SociaLite** (ottimizzato per problemi su grafi ed usato per lavorare su reti sociali), **BigDatalog** (utilizzato per Big Data Analytics) ed altri.

Mostriamo come implementazione nuovamente ABCDatalog con i programmi P_4 e P_3 :

Program:

```
person(a).
guilty(X) :- person(X), crime(X).
innocent(X) :- person(X), not guilty(X).
```

^ v ...

Load

Query: innocent(a) Run

Results:

```
innocent(a)
```

Program:

```
person(a).
guilty(X) :- person(X), crime(X).
innocent(X) :- person(X), not guilty(X).
```

^ v ...

Load

Query: guilty(a) Run

Results:

Program:

```
person(a).
guilty(X) :- person(X), not innocent(X).
innocent(X) :- person(X), not guilty(X).
```

^ v ...

Load

Query: Run

Results:

```
Error validating program: Program cannot be stratified.
```

Well-Founded Semantics

Mostriamo ora una diversa semantica che non restringa la classe di programmi su cui è applicabile mantenendo però l'unicità del modello canonico.

Il costo di questa generalizzazione è il passaggio ad una **logica a 3 valori** dove ogni atomo può essere o vero o falso o indefinito. L'idea è quella di far «collassare» tutti i modelli fondati in un singolo **well-founded model**, un'interpretazione composta da due insiemi T ed F .

L'insieme T contiene tutti gli atomi presenti in ogni modello fondato del programma P mentre F quelli non contenuti in nessun modello fondato. Questi due sottintendono un terzo insieme U degli atomi per cui esiste almeno un modello fondato che li contiene ed uno che non lo fa.

Dato un programma P , definiamo un suo well-founded model '**totale**' quando $T \cup F = HB(P)$ mentre lo chiamiamo '**parziale**' altrimenti. Quando il modello è totale l'insieme U è vuoto e ciò accade solo quando esiste un unico modello fondato.

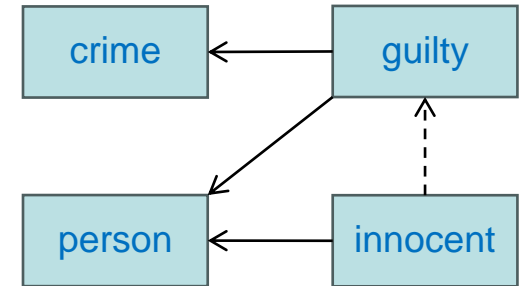
Well-Founded Semantics – Esempio 1

Esempio: riprendiamo il programma P_4

$\text{person}(a)$

$\text{guilty}(X) \leftarrow \text{person}(X), \text{crime}(X)$

$\text{innocent}(X) \leftarrow \text{person}(X), \text{not guilty}(X)$



Lo abbiamo già studiato: è stratificabile ed il modello perfetto $M_p = \{\text{person}(a), \text{innocent}(a)\}$ è l'unico modello fondato. Per quanto detto nella slide precedente il well-founded model è $M_w = \{T = \{\text{person}(a), \text{innocent}(a)\}, F = \{\text{crime}(a), \text{guilty}(a)\}\}$. Perciò $U = \emptyset$ e quindi M_w è totale.

In generale per programmi stratificabili poiché il modello fondato è unico ed è il modello perfetto, il well-founded model è totale. Modello perfetto e well-founded indicano la stessa situazione del mondo.

Well-Founded Semantics – Esempio 2

Esempio: P_5

$\text{guilty}(a) \leftarrow \text{not guilty}(b)$



Questo programma non è stratificabile per via di una ciclicità con negazione nel grafo delle dipendenze, il modello perfetto non esiste. Esistono due modelli minimi per P_5 :

$M_1 = \{\text{guilty}(a)\}$

$M_2 = \{\text{guilty}(b)\}$

Di questi solo M_1 è fondato, riusciamo perciò a trovare il well-founded model totale $M_W = \{T = \{\text{guilty}(a)\}, F = \{\text{guilty}(b)\}\}$.

Esistono dei programmi non stratificabili ma che per via dell'unicità del modello fondato risultano comunque avere well-founded model totale. Notiamo quindi che programmi stratificabili hanno un unico modello fondato ma il viceversa non vale.

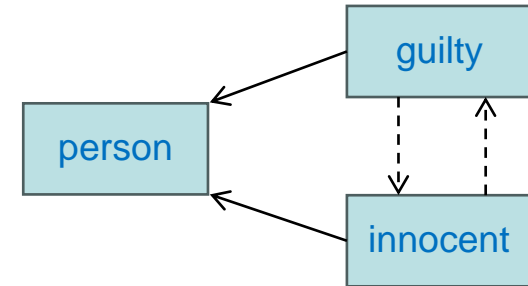
Well-Founded Semantics – Esempio 3

Esempio: riprendiamo il programma P_3

$\text{person}(a)$

$\text{guilty}(X) \leftarrow \text{person}(X), \text{not innocent}(X)$

$\text{innocent}(X) \leftarrow \text{person}(X), \text{not guilty}(X)$



Avevamo già notato come questo programma non fosse stratificabile per via di una ciclicità con negazione nel grafo delle dipendenze. Esistono due modelli minimi per P_3 e questi sono entrambi modelli fondati:

$M_1 = \{\text{person}(a), \text{guilty}(a)\}$

$M_2 = \{\text{person}(a), \text{innocent}(a)\}$

Data l'esistenza di più modelli fondati il well-founded model non è totale, $M_W = \{T = \{\text{person}(a)\}, F = \emptyset\}$ che sottintende $U = \{\text{guilty}(a), \text{innocent}(a)\}$.

Esistono programmi non stratificabili per cui non esiste well-founded model totale ma solo parziale.

Riduzione di Gelfond-Lifschitz

Definiamo la riduzione di Gelfond-Lifschitz (o **GL-reduct**) di un programma P rispetto ad una interpretazione I come il programma $P_I = \text{reduct}_P(I)$ ottenuto applicando i seguenti passaggi a $\text{ground}(P)$:

- Per ogni atomo $a \in I$ rimuovi le regole con il literal negativo $\text{not } a$ nel body
- Per ogni regola rimasta rimuovi il literal negativo $\text{not } a$ per ogni $a \notin I$

Intuitivamente stiamo «applicando» l'interpretazione al programma per sostituirne i literal negativi con il valore che assumono secondo l'interpretazione, ottenendo così un Positive Logic Program.

Esempio: riprendiamo il programma P_3 e sostituiamo ad ogni regola non-ground r il suo $\text{ground}(r)$

$\text{person}(a)$

$\text{guilty}(a) \leftarrow \text{person}(a), \text{not } \text{innocent}(a)$

$\text{innocent}(a) \leftarrow \text{person}(a), \text{not } \text{guilty}(a)$

$P_{3,I} :$

$\text{person}(a)$

$\text{innocent}(a) \leftarrow \text{person}(a)$

$I = \{\text{person}(a), \text{innocent}(a)\}$

Algoritmo di computazione del Well-Founded Model

Per il calcolo del well-founded model di un programma P viene utilizzato il seguente algoritmo, detto **Alternating Fixpoint**, che permette di evitare il calcolo di tutti i modelli fondati per P :

- Partiamo dall'interpretazione vuota $I_0 = \emptyset$ e calcoliamo il programma ridotto $P_0 = \text{reduct}_P(I_0)$
- Calcoliamo il modello minimo di P_0 che chiamiamo I_1 ed il nuovo programma ridotto $P_1 = \text{reduct}_P(I_1)$
- Iteriamo il passo precedente, calcolando da P_i il modello minimo I_{i+1} e da questo il programma ridotto P_{i+1} finché sia le interpretazioni pari I_{2k} sia quelle dispari I_{2k+1} non hanno raggiunto un punto fisso (ovvero finché l'algoritmo non **oscilla tra due punti fissi**).
- Costruiamo il well-founded model ponendo in T gli atomi presenti in entrambi i punti fissi ed in F gli atomi di $HB(P)$ non presenti in nessuno dei due (e quindi in U tutti gli altri).

Osserviamo che la successione di interpretazioni pari I_{2k} è una successione non decrescente di insiemi (ogni insieme contiene non strettamente il precedente) mentre la successione di interpretazioni dispari I_{2k+1} è una successione non crescente di insiemi (ogni insieme è contenuto non strettamente nel precedente).

Well-founded Datalog

Implementazioni di linguaggi logici che seguono well-founded semantics sono meno comuni rispetto a quelli stratificati.

Presentiamo **XSB**, un «dialetto» del linguaggio Prolog e che supporta parzialmente anche HiLog, una estensione di Prolog che permette di usare logiche di grado superiore. Il suo motore di inferenza è chiamato SLG-WAM.

La caratteristica principale di XSB è l'utilizzo del paradigma di **Tabled Logic Programs**, la cui idea di base è il salvare in una tabella i subgoals (infatti parliamo di un approccio top-down) per evitare di ricalcolarli nel caso si presentino più volte. Questo ragionamento esteso alla negazione permette di ritardare la valutazione di un atomo durante il processo di riduzione del programma.

Le clausole non ancora definite quando la riduzione raggiunge il punto fisso saranno viste come «undefined».

```
[xsb_configuration loaded]
[sysinitrc loaded]
[xsbbat loaded]

XSB Version 5.0-beta (Green Tea) of May 15, 2022
[x64-pc-windows; mode: optimal; engine: slg-wam; scheduling: local]
[Build date: 2022-05-10]

| ?- [programma_P3].
[Compiling .\programma_P3]
[programma_P3 compiled, cpu time used: 0.063 seconds]
[programma_P3 loaded]

yes
| ?- person(a).

yes
| ?- innocent(a).

undefined
| ?- guilty(a).

undefined
```

Computazione del Well-Founded Model - Criticità

Consideriamo il seguente programma:

person(a)

guilty(a) \leftarrow person(a), *not* innocent(a)

innocent(a) \leftarrow person(a), *not* guilty(a)

accused(a) \leftarrow person(a), innocent(a)

accused(a) \leftarrow person(a), guilty(a)

Applicando l'algoritmo per il calcolo del well-founded model otteniamo:

$I_0 = \{ \}$ $I_1 = \{ \text{person(a)}, \text{innocent(a)}, \text{guilty(a)}, \text{accused(a)} \}$

$I_2 = \{ \text{person(a)} \}$ $I_3 = \{ \text{person(a)}, \text{innocent(a)}, \text{guilty(a)}, \text{accused(a)} \}$

$I_4 = \{ \text{person(a)} \}$

Abbiamo quindi che come well-founded model $M_W = \{ T = \{ \text{person(a)} \}, F = \emptyset \}$. Se invece cerchiamo i modelli fondati troviamo $M_1 = \{ \text{person(a)}, \text{innocent(a)}, \text{accused(a)} \}$ ed $M_2 = \{ \text{person(a)}, \text{guilty(a)}, \text{accused(a)} \}$ che portano a $M_W' = \{ T = \{ \text{person(a)}, \text{accused(a)} \}, F = \emptyset \}$.

Computazione del Well-Founded Model - Criticità

Osserviamo che se vale $person(a)$ allora affinché l'interpretazione sia un modello deve valere almeno uno tra $innocent(a)$ e $guilty(a)$, quindi $accused(a)$ sarà vero per ogni modello. Potremmo vedere $accused(a)$ come direttamente inferito da $person(a)$.

L'algoritmo di computazione descritto non è in grado di effettuare questo ragionamento e vede $accused(a)$ come inferibile da due body tali che nessuno dei due sia vero per ogni modello fondato, ciò che gli manca è capire che **sempre** (per ogni modello fondato) **almeno uno dei due è vero**.

Osserviamo che tale criticità è presente anche in motori di inferenza per well-founded model semantics come SLG-WAM utilizzato in precedenza.

```
[xsb_configuration loaded]
[sysinitrc loaded]
[xsbbrat loaded]

XSB Version 5.0-beta (Green Tea) of May 15, 2022
[x64-pc-windows; mode: optimal; engine: slg-wam; scheduling: local]
[Build date: 2022-05-10]

| ?- [programma_criticita].
[Compiling .\programma_criticita]
[programma_criticita compiled, cpu time used: 0.079 seconds]
[programma_criticita loaded]

yes
| ?- accused(a).

undefined
```

Stable model semantics

Partendo sempre dalla riduzione di Gelfond-Lifschitz definiamo la Stable model semantics, una semantica di uso più generale rispetto a quella basata sul modello perfetto ma che invece di individuare un unico modello canonico definisce una collezione di modelli che chiamiamo **modelli stabili**.

Una interpretazione I per un programma P è un modello stabile quando il modello minimo del Positive Logic Program $reduct_P(I)$ è la stessa I .

Esempio: riprendiamo la riduzione di P_3 per l'interpretazione $I = \{\text{person}(a), \text{innocent}(a)\}$

$P_{3,I} :$

$\text{person}(a)$

$\text{innocent}(a) \leftarrow \text{person}(a)$

$I' = \{\text{person}(a), \text{innocent}(a)\}$ è il modello minimo per $P_{3,I}$ ed è uguale all'interpretazione I usata per la riduzione. I è quindi un modello stabile per P_3 .

Stable model semantics – Esempi

Consideriamo altre interpretazioni per lo stesso programma P_3

$$I_2 = \{\text{person}(a), \text{innocent}(a), \text{guilty}(a)\}$$

$$P_{3,I_2} : \quad \text{person}(a)$$

$$I_2' = \{\text{person}(a)\} \neq I_2$$

$$I_4 = \{\text{person}(a)\}$$

$$P_{3,I_4} : \quad \text{person}(a)$$

$$\text{guilty}(a) \leftarrow \text{person}(a)$$

$$\text{innocent}(a) \leftarrow \text{person}(a)$$

$$I_4' = \{\text{person}(a), \text{guilty}(a), \text{innocent}(a)\} \neq I_4$$

$$I_3 = \{\text{person}(a), \text{guilty}(a)\}$$

$$P_{3,I_3} : \quad \text{person}(a)$$

$$\text{guilty}(a) \leftarrow \text{person}(a)$$

$$I_3' = \{\text{person}(a), \text{guilty}(a)\} = I_3$$

Osserviamo che tutte le interpretazioni che non contengono $\text{person}(a)$ sicuramente non sono modelli.

Possiamo affermare che i modelli stabili di P_3 sono $I = \{\text{person}(a), \text{innocent}(a)\}$ e $I_3 = \{\text{person}(a), \text{guilty}(a)\}$.

Stable model semantics – Osservazioni

- Ogni modello stabile di un programma è un modello minimo di quel programma.
- Quando il programma è stratificabile (e quindi anche quando è un Positive Logic Program) il modello stabile è unico ed è uguale al modello perfetto, che a sua volta era analogo al well-founded model totale.
- Un modello di un programma è stabile sse il suo insieme infondato è vuoto, ovvero per come lo abbiamo definito è un modello fondato. Perciò i modelli presi in considerazione dalla well-founded semantics e dalla stable model semantics sono gli stessi.
- Il well-founded model è totale per un programma sse esiste un unico modello stabile.
- Poiché può esistere well-founded model totale per programmi non stratificabili l'esistenza di un unico modello stabile non implica l'esistenza del modello perfetto e quindi la stratificabilità del programma.

Interrogazioni per una collezione di modelli

La semantica del modello stabile attribuisce ad un programma non più un significato univoco ma una lista di scenari plausibili.

Quando facciamo una interrogazione riguardo il mondo modellizzato da un programma di questo tipo ci dobbiamo domandare come gestire le discrepanze tra i diversi modelli. Questo porta alla nascita di due tipi di ragionamento con cui interpretare le query ed i cui significati coincidono quando il modello canonico è unico:

- **Ragionamento cauto**: una query è vera quando lo è per ogni modello nella collezione
- **Ragionamento audace**: una query è vera quando lo è per almeno un modello nella collezione

Entrambi i ragionamenti sono **non-monotoni**: l'aggiunta di una ulteriore clausola nel programma potrebbe portare la conclusione da vera a falsa, perfino se la clausola aggiunta era conseguenza del programma.

Esempio: $P_6 = \{p(b) \leftarrow \text{not } p(c) ; p(c) \leftarrow \text{not } p(b) ; p(a) \leftarrow \text{not } p(a) ; p(a) \leftarrow p(b)\}$ ha come unico modello stabile $M_1 = \{p(a), p(b)\}$ per cui $p(a)$ e $p(b)$ sono conseguenze caute, tuttavia il programma $P_6 \cup \{p(a)\}$ ha come ulteriore modello stabile $M_2 = \{p(a), p(c)\}$ e ciò rende $p(b)$ non più una conseguenza cauta.

Nessun modello fondato

Abbiamo visto che ogni Normal Logic Program ha almeno un modello, tuttavia esistono Normal Logic Programs per i quali non esistono modelli stabili (e quindi nemmeno un well-founded model totale o parziale).

Questo significa che non sempre le semantiche che abbiamo analizzato sono in grado di assegnare un significato ad un Normal Logic Program.

Esempio: consideriamo il seguente programma P_7

$p(a) \leftarrow \text{not } p(a)$

L'unico modello di questo programma è $M = \{p(a)\}$. Per tale modello l'atomo $p(a)$ risulta infondato e quindi il modello non è stabile.

Stable model Datalog

L'**Answer Set Programming** (ASP) è una sottocategoria della programmazione logica basata sulla semantica del modello stabile. La sintassi comunemente accettata per l'ASP è detta AnsProlog, è un sottoinsieme di Prolog e contiene interamente Datalog.

Per questo non si trovano implementazioni di Datalog con negazione che utilizzano la semantica del modello stabile ma si possono utilizzare implementazioni di ASP, dette **Answer Set Solvers**, per lo stesso scopo.

Gli Answer Set Solver lavorano su programmi privi di variabili, hanno quindi bisogno di un «**grounder**» che converta questi programmi esplicitando tutte le regole ground implicate dalle regole con variabili.

Smodels è un Answer Set Solver creato per lavorare su programmi groundizzati da un altro programma, chiamato **Lparse**. E' la sintassi accettata da Lparse ad aver originariamente definito AnsProlog.

```
D:\smodels>lparse program | smodels 2
smodels version 2.26. Reading...done
Answer: 1
Stable Model: innocent(a) person(a)
Answer: 2
Stable Model: guilty(a) person(a)
True
Duration 0.000
Number of choice points: 1
Number of wrong choices: 1
Number of atoms: 3
Number of rules: 3
Number of picked atoms: 3
Number of forced atoms: 0
Number of truth assignments: 8
Size of searchspace (removed): 2 (0)
```

Stable model Datalog

Clingo è un altro ASP system per groundizzare e risolvere programmi logici. Il suo grounder è chiamato **Gringo** ed il suo Answer Set Solver **Clasp**.

```
1 % instance
2 person(a).
3
4 % encoding
5 innocent(X) :- person(X), not guilty(X).
6 guilty(X) :- person(X), not innocent(X).
```

Configuration: reasoning mode enumerate all ▼ ☐ project ☐ statistics

▶ Run!

```
clingo version 5.7.0
Reading from stdin
Solving...
Answer: 1
person(a) guilty(a)
Answer: 2
person(a) innocent(a)
SATISFIABLE

Models      : 2
Calls       : 1
Time        : 0.000s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

Extended Logic Programs (ELP)

Possiamo estendere ulteriormente i Normal Logic Programs tramite l'utilizzo della negazione forte. Definiamo un Extended Logic Program come un insieme finito di clausole del tipo:

$$a \leftarrow b_1, b_2, \dots, b_n, \text{not } c_1, \text{not } c_2, \dots, \text{not } c_m \quad (n, m \geq 0)$$

Dove $a, b_1, b_2, \dots, b_n, c_1, c_2, \dots, c_m$ sono atomi o negazioni forti di atomi in un linguaggio di logica del primo ordine.

Esempio:

$$\text{vola}(X) \leftarrow \text{uccello}(X), \text{not } \neg \text{vola}(X)$$

Questa clausola combina l'utilizzo di negazione forte e debole per indicare che «se non specificato altrimenti, se un elemento è un uccello allora vola»

Semantica di ELP tramite riduzione ad NLP

Definiamo una semantica per Extended Logic Programs tramite una riduzione ai Normal Logic Programs ottenuta con i seguenti passaggi:

- Per ogni predicato p di cui viene utilizzata la negazione forte definiamo un predicato dal nome « $\neg p$ »
- Definiamo un nuovo programma P' (che sarà un Normal Logic Program) sostituendo le negazioni forti di p con il predicato $\neg p$ ed aggiungendo per ogni nuovo predicato $\neg p$ la seguente clausola:
$$\text{falsity} \leftarrow \text{not falsity}, p(X), \neg p(X)$$
- I modelli scelti dalla semantica sono l'insieme dei modelli stabili di P' .

E' sempre possibile applicare questa semantica ad un ELP perciò l'estensione ad Extended Logic Programs non aumenta l'espressività dei programmi ma rende **più immediato tradurre concetti del linguaggio naturale**.

Ulteriori Estensioni

I **vincoli di integrità** sono delle regole senza la testa tali che se il body è verificato da una interpretazione, questa non è ammissibile. Come i goal esprimono l'opposto del loro body ma a differenza di questi possono appartenere al programma.

Esempio:

$[False] \leftarrow \text{guilty}(a), \text{guilty}(b)$ esprime che «a» e «b» non possono essere entrambi colpevoli

La **disgiunzione nella testa** permette di esprimere conoscenza indefinita ed aggiungere non-determinismo in un programma logico.

Esempio:

$\text{guilty}(a) \vee \text{guilty}(b) \leftarrow [True]$ esprime che almeno uno tra «a» e «b» è colpevole

$\text{innocent}(X) \vee \text{guilty}(X) \leftarrow \text{person}(X)$ esprime che ogni persona è innocente o colpevole

I programmi del tipo 'NLP + vincoli di integrità' possono essere ridotti a NLP in modo analogo agli ELP ([nel nostro esempio: falsity \$\leftarrow\$ not falsity, guilty\(a\), guilty\(b\)](#)). Invece quelli del tipo 'NLP + disgiunzioni nelle teste', detti Disjunctive Logic Programs (DLP) sono strettamente più espressivi degli NLP.

Riferimenti

- Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner: Answer Set Programming: A Primer
- Stefania Costantini: Answer Set Programming: an Introduction
- Robert Kowalski: History of Logic Programming
- Keith L. Clark: Negation As Failure
- Allen Van Gelder, Kenneth A. Ross, John S. Schlipf: The Well-Founded Semantics for General Logic Programs
- Aaron Bembenek, Stephen Chong, Marco Gaboardi: ABCDatalog <https://harvardpl.github.io/AbcDatalog/>
- Paraschos Koutris: Lecture 8: Datalog: Evaluation
- Bas Ketsman, Paraschos Koutris: Modern Datalog Engines
- XSB <https://xsb.sourceforge.net/>
- Yi Zhou: First-Order Disjunctive Logic Programming vs Normal Logic Programming
- Smodels <http://www.tcs.hut.fi/Software/smodels/>
- Clingo <https://potassco.org/clingo/>