Note sui programmi logici e loro applicazione nella specifica di politiche di sicurezza

P.A. Bonatti

5 dicembre 2013, Versione 5

1 Introduzione

Nell'area della sicurezza sono stati introdotte numerose politiche e modelli diversi, sulla spinta delle esigenze delle diverse applicazioni. È difficile dire se le politiche introdotte sinora coprano tutti i casi possibili, o se in futuro si riveleranno necessari nuovi modelli, come conseguenza degli sviluppi delle metodologie e delle tecnologie informatiche. Inoltre politiche e modelli diversi devono frequentemente convivere all'interno di organizzazioni di una certa dimensione. Dipartimenti diversi, dati relativi a diversi domini, possono richiedere l'applicazione simultanea di politiche diverse, talora opposte tra loro.

Questo porta a requisiti di forte flessibilità cui però non si vuole rispondere con ambienti di specifica di politiche basati su linguaggi di programmazione tradizionali, che richiedono personale specializzato nella programmazione e producono codice che per la sua stessa natura è difficile da controllare e manutenere.

Per ottenere la flessibilità desiderata senza rinunciare ai benefici della specifica dichiarativa (non procedurale) di politiche, si presta oggi un sempre maggiore interesse ai linguaggi logici.

In queste note forniamo una breve introduzione ai linguaggi logici che stanno alla base delle proposte più espressive di linguaggi di specifica di politiche. In quest'ambito ci si restringe al cosiddetto frammento DATALOG dei programmi logici con negazione, già utilizzato in passato per modellare basi di dati deduttive, e caratterizzato da buone proprietà computazionali, prima tra le quali la decidibilità.

2 Programmi logici positivi

2.1 Sintassi

Il linguaggio di un programma logico DATALOG si basa su un insieme finito di *simboli di predicato*, un insieme finito di *costanti* e un insieme illimitato di

variabili.[1]

Adottiamo la convenzione di PROLOG e denotiamo le costanti mediante identificatori che iniziano con una lettera minuscola, mentre le variabili iniziano con una lettera maiuscola.

Un *termine* è una costante o una variabile.

Un *atomo* è una espressione della forma $p(t_1, \ldots, t_n)$ $(n \geq 0)$, dove p è un simbolo di predicato e i t_j $(1 \leq i \leq n)$ sono termini. Ogni atomo senza variabili rappresenta una affermazione che può essere vera o falsa.

Esempio 1 Possiamo codificare con un atomo $\texttt{aut}(\texttt{ann}, \texttt{read}, \texttt{doc1})$ l'affermazione che Ann è autorizzata a leggere il file doc1, cioè una autorizzazione. Tale affermazione può essere vera o falsa, a seconda che l'autorizzazione sia concessa o meno. □

Un *programma logico positivo* è un insieme finito di regole della forma:

$$A \leftarrow B_1, \ldots, B_m$$

dove A e i B_i $(1 \leq i \leq m)$ sono atomi. L'atomo A viene detto *testa* della regola, mentre l'insieme $\{B_1, \ldots, B_m\}$ viene detto *corpo* della regola. Le regole con corpo vuoto $(m = 0)$ verranno dette *fatti* ed identificate con l'atomo nella testa.

Il significato intuitivo di una regola con variabili $X_1, \ldots, X_k$ è: *per ogni* $X_1, \ldots, X_k$, *se* $B_1, \ldots, B_m$ *sono veri, allora* A *è vero.*

Esempio 2 Spesso l'autorizzazione alla scrittura è ritenuta più forte della lettura, e si vuole che la prima implichi la seconda. Questa politica si può formulare con la regola

$$\texttt{aut}(\texttt{Sogg}, \texttt{read}, \texttt{Ogg}) \leftarrow \texttt{aut}(\texttt{Sogg}, \texttt{write}, \texttt{Ogg})\,.$$

Qui `Sogg` e `Ogg` sono variabili, perciò si intende che la regola si applica a tutti i soggetti e a tutti gli oggetti. □

Ogni programma logico positivo può essere visto come una abbreviazione della sua *istanziazione ground*, definita nel seguito. Una espressione (termine, atomo o regola) è *ground* se non contiene variabili. Una *istanza ground* di un atomo o di una regola si ottiene sostituendo una costante a ciascuna variabile (avendo cura di sostituire diverse occorrenze della stessa variabile con la stessa costante, mentre diverse variabili possono essere sostituite con la stessa costante). Infine l'istanziazione ground di un programma logico P, denotata da $\mathsf{Ground}(P)$, è l'insieme di tutte le possibili istanze ground delle sue regole (coerentemente con l'implicita quantificazione universale delle variabili).

L'identificazione dei programmi logici con la loro istanziazione ground ci permetterà nei prossimi capitoli di formulare e studiare la semantica dei programmi logici restringendoci ai soli programmi ground, senza perdere generalità.

[1] Quando non ci restringiamo al frammento DATALOG, oltre ai simboli di costante, il linguaggio può comprendere simboli di funzione n-ari.

Nel seguito, se non altrimenti specificato, assumiamo che il linguaggio di un programma logico consista dei simboli di predicato, delle costanti e delle variabili che compaiono nel programma.[2]

2.2 Semantica di punto fisso

Come si calcolano le conseguenze delle regole di un programma positivo ground P? Un possibile algoritmo consiste nel partire da un insieme vuoto, generare al primo passo i fatti contenuti in P (cioè quegli atomi che sono veri senza alcun prerequisito, visto che sono in testa ad una regola il cui corpo è banalmente soddisfatto essendo vuoto), poi applicare al passo 2 tutte le regole il cui corpo si è rivelato vero al passo 1, e così via.

La definizione formale si basa sull'*operatore delle conseguenze immediate* di P, che trasforma insiemi di atomi ground in altri insiemi dello stesso tipo:

$$T_P(I) = \{A \mid (A \leftarrow B_1, \ldots, B_m) \in P \text{ and } \{B_1, \ldots, B_m\} \subseteq I\}$$

dove I rappresenta l'insieme degli atomi già dimostrati veri al passo precedente. Intuitivamente, T_P applica in parallelo tutte le regole il cui corpo è soddisfatto. L'operatore T_P rappresenta il calcolo del singolo passo, e va iterato più volte, come già detto. L'applicazione di T_P n volte a partire da una interpretazione iniziale I si denota con $T_P^n(I)$ ed è definita ricorsivamente come segue:

$$\begin{aligned} T_P^0(I) &= I\,, \\ T_P^n(I) &= T_P(T_P^{n-1}(I)) \quad (n > 0). \end{aligned}$$

Ne segue che:

$$T_P^n(I) = \underbrace{T_P(T_P(\ldots T_P}_{n}(I)\ldots))\,.$$

La sequenza $T_P^0(\emptyset), T_P^1(\emptyset), \ldots, T_P^i(\emptyset), \ldots$ è crescente e raggiunge un punto di saturazione, ovvero un indice k tale che $T_P^k(\emptyset) = T_P^{k+1}(\emptyset) = T_P^{k+2}(\emptyset) = \ldots$; chiaramente questo punto di saturazione è un *punto fisso*[3] di T_P, in quanto

$$T_P^k(\emptyset) = T_P^{k+1}(\emptyset) = T_P(T_P^k(\emptyset))\,.$$

Per mostrare che iterando T_P a partire dall'insieme vuoto si ottiene una sequenza monotòna di questo tipo, iniziamo col dimostrare che T_P è *monotòno* rispetto al suo input, cioè per tutte le interpretazioni di Herbrand I e J, $I \subseteq J$ implica che $T_P(I) \subseteq T_P(J)$.

Lemma 1 *Per tutte le interpretazioni di Herbrand I e J, $I \subseteq J$ implica che $T_P(I) \subseteq T_P(J)$.*

[2] Si noti che in generale i simboli del linguaggio possono essere un soprainsieme stretto di quelli qui specificati.

[3] I punti fissi di T_P sono quegli insiemi I tali che $I = T_P(I)$.

Dimostrazione: Supponiamo che $I \subseteq J$. Sia A un qualunque elemento di $T_P(I)$. Dobbiamo dimostrare che $A \in T_P(J)$. Se $A \in T_P(I)$, allora per definizione esiste una regola $A \leftarrow B_1, \ldots, B_n$ in P tale che $\{B_1, \ldots, B_n\} \subseteq I$. Ne segue (per l'ipotesi che $I \subseteq J$) che $\{B_1, \ldots, B_n\} \subseteq J$. Dunque per definizione di T_P abbiamo $A \in T_P(J)$. ∎

Ora possiamo dimostriare che le iterazioni di T_P a partire dall'insieme vuoto generano insiemi via via crescenti, cioè $\emptyset \subseteq T_P(\emptyset) \subseteq \cdots \subseteq T_P^i(\emptyset) \subseteq T_P^{i+1}(\emptyset) \subseteq \cdots$.

Lemma 2 *Per ogni $i \geq 0$, $T_P^i(\emptyset) \subseteq T_P^{i+1}(\emptyset)$.*

Dimostrazione: Per induzione su i. Caso base ($i = 0$): $T_P^0(\emptyset) = \emptyset \subseteq T_P^1(\emptyset)$. Passo di induzione ($i = 0$): Supponiamo per ipotesi di induzione che

$$T_P^{i-1}(\emptyset) \subseteq T_P^i(\emptyset).$$

ne segue, per il Lemma 1, che $T_P(T_P^{i-1}(\emptyset)) \subseteq T_P(T_P^i(\emptyset))$, che per definizione è equivalente a $T_P^i(\emptyset) \subseteq T_P^{i+1}(\emptyset)$. ∎

Infine basta notare che l'insieme di tutte le interpretazioni di Herbrand di un programma logico (finito) DATALOG è a sua volta finito (poichè l'insieme degli atomi ground di tali programmi è finito) per vedere che la sequenza $\langle T_P^i(\emptyset)\rangle_{i\geq 0}$ non può crescere all'infinito. Nel caso estremo, ad ogni applicazione di T_P prima della saturazione si aggiunge un solo atomo preso dalla testa di un'unica regola di P, perciò deve esistere $k \leq |P|$ tale che $T_P^k(\emptyset)$ è un *punto fisso* di T_P, vale a dire

$$T_P(T_P^k(\emptyset)) = T_P^k(\emptyset)\,.$$

Per ogni programma P denoteremo con M_P il corrispondente punto fisso $T_P^k(\emptyset)$ così ottenuto. Nei prossimi capitoli proveremo che M_P è l'*unico minimo* punto fisso di T_P (secondo l'ordinamento dato da $\subseteq$).

Esempio 3 Riprendiamo la regola introdotta nell'esempio 3 e consideriamo il programma

$$\begin{aligned} P \;=\; \{\, &\mathtt{aut(ann, read, doc1)},\ \mathtt{aut(bob, write, doc2)},\\ &\mathtt{aut(S, read, O)} \leftarrow \mathtt{aut(S, write, O)}\}\,. \end{aligned}$$

Qui $\mathsf{Ground}(P)$ comprende i fatti $\mathtt{aut(ann, read, doc1)}$ e $\mathtt{aut(bob, write, doc2)}$ nonchè tutte le istanze della regola dove S e O sono sostituite con le costanti `ann`, `bob`, `read`, `write`, `doc1`, `doc2` in tutti i modi possibili.[4] Le conseguenze di P si trovano calcolando la sequenza $\{T^i_{\mathsf{Ground}(P)}(\emptyset)\}_{i\geq 0}$:

$$\begin{aligned} T^0_{\mathsf{Ground}(P)}(\emptyset) &= \emptyset\\ T^1_{\mathsf{Ground}(P)}(\emptyset) &= \{\mathtt{aut(ann, read, doc1)},\ \mathtt{aut(bob, write, doc2)}\}\\ T^2_{\mathsf{Ground}(P)}(\emptyset) &= T^1_{\mathsf{Ground}(P)}(\emptyset) \cup \{\mathtt{aut(bob, read, doc2)}\}\\ T^3_{\mathsf{Ground}(P)}(\emptyset) &= T^2_{\mathsf{Ground}(P)}(\emptyset)\,. \end{aligned}$$

[4] Si noti che le istanze ground comprendono anche regole "non desiderabili", ad esempio dove `S` è legata a `read` o `write`. Tuttavia, se i fatti sono codificati corrrettamente, tali regole rimangono inattive perchè il loro corpo non viene soddisfatto.

Per cui $M_{\mathsf{Ground}(P)} = T^2_{\mathsf{Ground}(P)}(\emptyset)$ e le conseguenze di P risultano essere gli atomi: $\mathtt{aut}(\mathtt{ann}, \mathtt{read}, \mathtt{doc1})$, $\mathtt{aut}(\mathtt{bob}, \mathtt{write}, \mathtt{doc2})$, $\mathtt{aut}(\mathtt{bob}, \mathtt{read}, \mathtt{doc2})$. □

Con opportuni algoritmi, il punto fisso M_P può essere calcolato in tempo $O(n \log n)$ [1].

Esercizio 1 *Rappresentare la politica no-read-up no-write-down con un programma logico positivo.*

2.3 Semantica operazionale

Volendo verificare se un particolare atomo A è una conseguenza di un programma positivo P senza generare l'intero punto fisso, possiamo adottare un sistema duale al procedimento introdotto nella sezione precedente. Possiamo cercare una regola $R \in \mathsf{Ground}(P)$ con A nella testa, e in caso positivo sostituire A con il corpo di R e procedere ricorsivamente cercando di dimostrare quest'ultimo.

Esempio 4 Riprendiamo il programma dell'esempio 3. Dato l'atomo ground $A = \mathtt{aut}(\mathtt{bob}, \mathtt{read}, \mathtt{doc2})$, otteniamo la sequenza seguente:

$\mathtt{aut}(\mathtt{bob}, \mathtt{read}, \mathtt{doc2})$	riscritto con la regola in:
$\mathtt{aut}(\mathtt{bob}, \mathtt{write}, \mathtt{doc2})$	riscritto col secondo fatto in:
□	che denota un "goal vuoto".

Non essendoci più nulla da dimostrare, qui cessa la derivazione, con successo. □

Questo procedimento si formalizza mediante la nozione di *SLD-derivazione ground*. Questa è una sequenza di *goals* $G_0, G_1, \ldots, G_k$, ciascuno dei quali è una sequenza finita di atomi (che si vogliono dimostrare). Il goal vuoto si denota con □. Ciascun elemento G_i con $1 \leq i \leq k-1$ deve soddisfare la seguente proprietà:

> se $G_i = A_1, \ldots, A_i, \ldots, A_n$, esistono un indice i e una regola $A_i \leftarrow B_1, \ldots, B_m$ in $\mathsf{Ground}(P)$ tali che
>
> $$G_{i+1} = A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n.$$

In altre parole, A_i viene "riscritto" col corpo della regola. La derivazione è di *successo* se $G_k = \Box$.

Ora un atomo A è conseguenza di P sse esiste (almeno) una SLD-derivazione ground di successo con $G_0 = A$.

Si può dimostrare che questa nozione di conseguenza è equivalente alla nozione di conseguenza basata sulla semantica di punto fisso. La nozione di SLD-derivazione specifica solo un procedimento diverso per calcolare la stessa cosa.

Il procedimento basato sulla nozione di SLD-derivazione, contrariamente alla semantica di punto fisso, è nondeterministico: occorre scegliere l'atomo A_i da riscrivere e la regola da applicare (più regole potrebbero avere la stessa testa A_i). Si può dimostrare che la scelta dell'indice i non influisce sul successo, mentre la scelta della regola ad ogni passo è chiaramente critica.

Esercizio 2 *Scrivere un programma e scegliere un goal che ha due derivazioni* complete*—cioè che non possono essere ulteriormente estese—di cui una sola di successo.*

2.4 Valutazione parziale

Dalla semantica operazionale risulta chiaramente che in ogni programma ground P, una qualunque regola $A \leftarrow B_1, \ldots, B_m$ può essere sostituita *senza cambiare le conseguenze di* P riscrivendo un qualunque atomo B_i nel corpo con tutte le regole di P che hanno B_i nella testa. Più precisamente, se tali regole sono $B_1 \leftarrow \beta_1, \ldots, B_1 \leftarrow \beta_k$ (dove i β_j abbreviano i corpi delle regole), allora $A \leftarrow B_1, \ldots, B_i, \ldots, B_m$ può essere rimpiazzata con

$$\begin{array}{lcl} A & \leftarrow & B_1, \ldots, B_{i-1}, \beta_1, B_{i+1}, \ldots, B_m \\ & \vdots & \\ A & \leftarrow & B_1, \ldots, B_{i-1}, \beta_k, B_{i+1}, \ldots, B_m \, . \end{array}$$

In particolare, se nessuna regola ha B_i nella testa, la regola $A \leftarrow B_1, \ldots, B_m$ non può essere mai applicata, e può essere semplicemente rimossa. Se invece l'unica regola con B_i in testa è un fatto (corpo vuoto), allora B_i viene semplicemente eliminato dal corpo, e la regola viene rimpiazzata con

$$A \leftarrow B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_m \, .$$

Una *valutazione parziale* di P è il risultato di una arbitraria serie di sostituzioni effettuate con le suddette modalità. La valutazione parziale preserva l'insieme delle conseguenze di P.

2.5 Semantica dichiarativa

Ogni regola $A \leftarrow B_1, \ldots, B_m$ può essere vista come una formula della logica del primo ordine, $A \leftarrow B_1 \wedge \ldots \wedge B_m$. Queste formule possono essere viste come un filtro sull'insieme di tutti i possibili *stati del mondo*; solo i mondi dove l'eventuale verità di $B_1, \ldots, B_m$ implica la verità di A sono compatibili con la regola. Intuitivamente, gli stati del mondo compatibili con tutte le regole di un programma P sono tutti e soli quelli ritenuti possibili da chi possiede la conoscenza rappresentata da P. I fatti che sono veri in tutti i possibili stati del mondo compatibili con P sono sicuramente veri (sempre che la conoscenza codificata in P sia corretta). Similmente, i fatti che sono falsi in tutti i possibili stati del mondo compatibili con P sono sicuramente falsi. Infine, quei fatti che

sono veri in certi stati del mondo compatibili con P e falsi in altri, sono fatti il cui valore di verità non è univocamente specificato da P (che pertanto, in generale, rappresenta un corpo "incompleto" o "parziale" di conoscenza).

Formalmente, gli stati del mondo possono essere rappresentati mediante funzioni, dette *interpretazioni di Herbrand*, che mappano ogni atomo ground su $\{0, 1\}$, dove 0 rappresenta il falso ed 1 il vero, identificando così i fatti veri e falsi in quel particolare stato del mondo. Poichè ciascuna interpretazione è, per la sua forma, la funzione caratteristica di un particolare insieme di atomi, identifichiamo tali funzioni con i corrispondenti insiemi di atomi ground. Quindi un'interpretazione di Herbrand è un insieme di atomi ground, che contiene tutti e soli gli atomi *veri* in quello stato del mondo.

Un'interpretatione I *soddisfa* una regola ground $A \leftarrow B_1, \ldots, B_m$ se $A \in I$ (cioè A è vero) oppure per qualche i $(1 \leq i \leq m)$ $B_i \notin I$. Si noti che la regola *non* è soddisfatta sse il corpo è soddisfatto mentre la testa non lo è, cioè $\{B_1, \ldots, B_m\} \subseteq I$ e $A \notin I$. Diciamo che I soddisfa una regola non-ground R se I soddisfa tutte le istanze ground di R. Infine I soddisfa un programma P se soddisfa tutte le sue regole. Si noti che I soddisfa P se e solo se I soddisfa $\mathsf{Ground}(P)$. Se I soddisfa una qualche espressione E (fatto, regola o programma) scriviamo $I \models E$. Se $I \models E$, diciamo anche che I è un *modello* di E. Intuitivamente, la relazione $\models$ formalizza la nozione di compatibilità menzionata all'inizio di questa sezione. I modelli di un programma P rappresentano gli stati del mondo compatibili con la conoscenza codificata in P.

Un atomo A è *conseguenza logica* di P sse A è soddisfatto da tutti i modelli di P. In questo caso scriviamo $P \models A$. Si noti che la nozione di conseguenza logica cattura i fatti sicuramente veri se la conoscenza (parziale) codificata in P è vera.

La semantica dichiarativa di P consiste proprio nell'insieme di conseguenze logiche di P, vale a dire, l'intersezione di tutti i modelli di P. Per i programmi positivi, tale intersezione coincide sempre con un particolare modello, cioè l'unico *modello minimo* di P. La minimalità è intesa rispetto alla relazione di inclusione insiemistica: un modello M di P è minimo se per ogni modello I di P, $I \subseteq M$ implica $I = M$.

Dimostriamo che ogni programma positivo P ha esattamente un modello minimo. Per vedere che P ha sempre almeno un modello, si noti che l'interpretazione di Herbrand che contiene tutti gli atomi ground è un modello di P, perchè sicuramente rende vere le teste di tutte le clausole. Per vedere che esiste un solo modello minimo usiamo il seguente lemma.

Lemma 3 *Se I e J sono modelli di P, allora $I \cap J$ è un modello di P.*

Dimostrazione: (Per assurdo) Supponiamo di no, e raggiungiamo una contraddizione. Siano I e J due arbitrari modelli di P. Se $I \cap J \not\models P$, allora esiste una istanza ground di una regola di P, $A \leftarrow B_1, \ldots, B_m$, tale che

$$\{B_1, \ldots, B_m\} \subseteq I \cap J\,, \tag{1}$$

$$A \notin I \cap J\,. \tag{2}$$

Da (1), abbiamo $\{B_1, \ldots, B_m\} \subseteq I$ e $\{B_1, \ldots, B_m\} \subseteq J$, quindi siccome I e J sono modelli di P (dunque della regola) dobbiamo avere $A \in I$ e $A \in J$, cioè $A \in I \cap J$. Ma questo contraddice (2). ∎

Ora, poichè i modelli di Herbrand di un programma DATALOG P sono in numero finito (2^a dove a è il numero di atomi ground nel linguaggio) è facile vedere per induzione sul numero di modelli che la loro intersezione è ancora un modello di P, utilizzando il Lemma 3. Ne segue che

Teorema 5 *Ogni programma* DATALOG *positivo ha esattamente un modello minimo.*

Chiaramente, poichè il modello minimo di P coincide con l'intersezione di tutti i suoi modelli, coincide anche con l'insieme delle conseguenze logiche di P (segue immediatamente dalle definizioni).

Notazione: il modello minimo di un programma P verrà denotato con $\mathsf{lm}(P)$.[5]

È importante osservare che tutte le semantiche fin qui introdotte sono solo le diverse facce di un'unica entità, nel senso che coincidono tutte. Formalmente:

Teorema 6 *Dato un qualunque programma logico positivo P, il minimo punto fisso di T_P, l'insieme degli atomi che posseggono una SLD-derivazione di successo da P, e il modello minimo di P coincidono.*

Siamo quindi liberi di scegliere di volta in volta la formalizzazione più conveniente.

Nel seguito dimostriamo parte del teorema qui sopra, e più precisamente la corrispondenza tra il modello minimo $\mathsf{lm}(P)$ (ovvero l'insieme delle conseguenze logiche atomiche ground di P) e il minimo punto fisso di T_P .

Ricordate che il minimo punto fisso M_P di T_P si ottiene iterando T_P a partire dall'insieme vuoto, pertanto $M_P = {}^k_P(\emptyset)$, per qualche intero $k \geq 0$.

Per mostrare che $M_P = \mathsf{lm}(P)$ (cioè che il minimo punto fisso di T_P contiene effettivamente tutti e soli gli atomi ground che sono conseguenze logiche di P) occorrono alcuni risultati intermedi.

Diciamo che una interpretazione di Herbrand I è un *pre-punto fisso* di T_P sse $T_P(I) \subseteq I$.

Lemma 4 *I è un modello di P sse I è un pre-punto fisso di T_P.*

Dimostrazione: ($\Rightarrow$) Supponiamo che I sia un modello di P ma per assurdo I non sia un pre-punto fisso di T_P, ovvero $T_P(I) \not\subseteq I$. Ciò implica che esiste un atomo A che appartiene a $T_P(I)$ ma non a I. Allora per definizione di T_P deve esistere una regola $A \leftarrow B_1, \ldots, B_n$ in P tale che $\{B_1, \ldots, B_n\} \subseteq I$. Questo implica che $I \models \{B_1, \ldots, B_n\}$, quindi $I \models A$ (altrimenti I non soddisferebbe la regola e pertanto non sarebbe un modello di P, in contraddizione con l'ipotesi iniziale). Ma allora $A \in I$ (una contraddizione). Questo dimostra il "solo se".

[5] Dall'abbreviazione di *least model*, che in inglese significa precisamente modello minimo.

($\Leftarrow$) Viceversa, supponiamo che I sia un pre-punto fisso di T_P ma, per assurdo, I non sia un modello di P. Questo significa che per qualche regola $A \leftarrow B_1, \ldots, B_n$ in P, abbiamo $I \models \{B_1, \ldots, B_n\}$ e $I \not\models A$ (cioè $A \notin I$). Per definizione di T_P ne segue che $A \in T_P(I)$, e siccome $A \notin I$, ne consegue che $T_P(I) \not\subseteq I$, il che contraddice l'ipotesi che I fosse un pre-punto fisso di T_P. ∎

Notate che i punti fissi di T_P sono anche pre-punti fissi, quindi il nostro punto fisso M_P, per il Lemma 4, è anche un modello di P. Il prossimo passo sarà dimostrare che è il modello *minimo* di P nell'ordinamento determinato dall'inclusione insiemistica.

Teorema 7 *M_P è il modello minimo di P.*

Dimostrazione: Sia I un qualunque modello di P. Dobbiamo dimostrare che $M_P \subseteq I$. A questo scopo, è sufficiente dimostrare che

$$\text{per ogni } i \geq 0,\ T_P^i(\emptyset) \subseteq I\,, \tag{3}$$

perchè siccome $M_P = T_P^k(\emptyset)$, per qualche $k \geq 0$, la (3) implica in particolare che $M_P \subseteq I$ (che è precisamente quanto dobbiamo dimostrare).

Dimostriamo (3) per induzione su i. Se $i = 0$ allora banalmente $T_P^0(\emptyset) = \emptyset \subseteq I$. Passiamo dunque al passo induttivo ($i > 0$) e assumiamo per ipotesi di induzione che

$$T_P^{i-1}(\emptyset) \subseteq I\,. \tag{4}$$

Poichè I è un modello di P, per il Lemma 4 abbiamo:

$$T_P(I) \subseteq I\,. \tag{5}$$

Pertanto, applicando T_P a entrambi i termini della (4) e sfruttando la monotonia di T_P e la (5) otteniamo

$$T_P^i(\emptyset) = T_P(T_P^{i-1}(\emptyset)) \subseteq T_P(I) \subseteq I\,. \tag{6}$$

Questo conclude la dimostrazione della (3), e con essa del teorema. ∎

Poichè i punti fissi di T_P, come dicevamo, sono particolari modelli di P, il teorema precedente ci dice anche che M_P è il *minimo punto fisso* di T_P.

Con il Teorema 7 è facile mostrare che le conseguenze logiche atomiche di P sono esattamente i membri di M_P:

Teorema 8 *$P \models A$ sse $A \in M_P$.*

Dimostrazione: Se $P \models A$, allora per definizione A è soddisfatto da tutti i modelli di P, compreso M_P, per cui $A \in M_P$.

Viceversa, se $A \in M_P$, allora, per il Teorema 7, A appartiene anche a tutti i modelli di P (che contengono M_P), quindi A è soddisfatto da tutti i modelli di P il che significa precisamente che A è una conseguenza logica di P. ∎

Il teorema qui sopra completa la dimostrazione di corrispondenza tra la semantica di punto fisso e quella dichiarativa. Conferma anche che la procedura

iterativa che genera la sequenza $\{T_P^i(\emptyset)\}_{i\geq 0}$ implementa correttamente il calcolo delle conseguenze logiche di P ed è *completa*, cioè non ne perde nessuna.

Un'ultima osservazione. Quando usiamo i programmi positivi come un linguaggio di specifica per politiche di sicurezza, i possibili stati del mondo sono in realtà i possibili insiemi di autorizzazioni che possiamo formulare. Adottando una politica P, intendiamo specificare che l'insieme di autorizzazioni da utilizzare per il controllo degli accessi è il modello minimo di P.

Si noti che P e i relativi meccanismi di valutazione (semantiche operazionale e di punto fisso) *non* realizzano di per se il controllo degli accessi, ma si limitano a dire qual'è l'insieme di autorizzazioni specificato concisamente e implicitamente da P. Possiamo usare, per esempio, la semantica di punto fisso per produrre tale insieme di autorizzazioni, per poi trasformarlo in (qualche implementazione di) una matrice di controllo degli accessi, che a quel punto può essere utilizzata direttamente dall'effettivo meccanismo di controllo degli accessi presente nel sistema. In altre parole, l'adozione di un linguaggio logico per la specifica di politiche non è in linea di principio in contrasto con l'utilizzo di meccanismi tradizionali per il controllo degli accessi, nè causa a priori problemi di efficienza, in quanto la parte di ragionamento sulla politica può essere effettuata off-line.

2.6 Limiti dei programmi positivi

Supponiamo di voler rappresentare una politica chiusa. Dovremmo codificare una regola R che—in qualche modo—dica:

$$\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},-) \leftarrow \text{“non è specificata l'autorizzazione } \texttt{aut}(\texttt{S},\texttt{A},\texttt{O},+)\text{”}.$$

Ci aspetteremmo inoltre che il programma $P = \{R\}$ dovrebbe avere tra le sue conseguenze logiche $\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},-)$, mentre $P' = P \cup \{\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},+)\}$ *non* dovrebbe avere $\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},-)$ come conseguenza logica.

Si noti che ciò implica che l'insieme delle conseguenze del programma *non* dovrebbe crescere monotonicamente al crescere del programma stesso. In questo esempio, l'inserimento di $\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},+)$ nel programma dovrebbe causare la sparizione dell'autorizzazione $\texttt{aut}(\texttt{S},\texttt{A},\texttt{O},-)$.

Tuttavia la semantica dei programmi positivi è monotòna (non decrescente). Questo si vede facilmente esaminando la semantica operazionale. Ogni SLD-derivazione possibile in P è anche possibile in tutte le estensioni $P' \supseteq P$, perchè si basa solo sulla *presenza* di opportune regole nel programma, mai sulla loro assenza. L'aggiunta di regole e fatti può solo aumentare l'insieme di SLD-derivazioni di successo.

Monotòne sono anche l'intera logica del primo ordine (di cui i programmi positivi sono un frammento), i sistemi modali classici, la logica intuizionista ecc. Ne segue che regole come quella richiesta per la politica chiusa non possono essere formulate in nessun modo con le logiche classiche, inclusi i programmi positivi.

Ci occorre dunque una logica *nonmonotòna*. La semantica dei modelli stabili che introdurremo nei prossimi capitoli ha questa caratteristica.

3 Programmi logici normali

Introduciamo un nuovo costrutto `not`, detto *negazione per fallimento*, con cui intendiamo denotare la *non derivabilità* di un atomo. Vogliamo cioè definire una nuova forma di conseguenza logica $\models_n$ tale che per ogni atomo ground A,

$$P \models_n \texttt{not}\, A \text{ se e solo se } P \not\models_n A\,. \tag{7}$$

Inoltre vogliamo che $\models_n$ estenda naturalmente la nozione classica di conseguenza logica, cioè vogliamo che se il nuovo costrutto non viene utilizzato in un programma logico P, allora P mantenga la propria semantica tradizionale. Formalmente, ciò significa che per ogni programma positivo P,

$$P \models_n A \text{ se e solo se } P \models A\,. \tag{8}$$

Sotto queste due condizioni, $\models_n$ deve essere necessariamente nonmonotòna. Si consideri ad esempio il programma $P = \emptyset$. Per soddisfare (8), occorre che $P \not\models_n A$ e dunque, per la (7), $P \models_n \texttt{not}\, A$. Se ora estendiamo il programma a $P' = \{A\}$, allora per soddisfare (8) e (7), bisogna che $P \models_n A$ e $P \not\models_n \texttt{not}\, A$. In questo esempio, al crescere di P, l'insieme delle conseguenze negative decresce, il che mostra che $\models_n$ è nonmonotona.

Procediamo ora con le definizioni formali. Un *programma logico normale* è un insieme finito di regole della forma:

$$A \leftarrow L_1, \ldots, L_m$$

dove A è un atomo e gli L_i sono *letterali*, vale a dire espressioni della forma B o $\texttt{not}\, B$, dove B è un atomo. Si parla rispettivamente di letterali *positivi* e *negativi*.

Nel contesto dei programmi logici normali, le interpretazioni di Herbrand I rappresentano i possibili insiemi di atomi ground *derivabili*. Di conseguenza, rispetto a I, i letterali negativi veri sono tutti e soli i $\texttt{not}\, B$ tali che $B \notin I$.

Naturalmente, dato un programma logico normale P, non tutti gli I catturano correttamente gli atomi derivabili da P. Un atomo, per essere derivabile (e appartenere ad I), dovrebbe essere una conseguenza delle regole di P e dei letterali $\texttt{not}\, B$ veri, cioè dovrebbe valere l'equazione:

$$I = \{A \mid P \cup \{\texttt{not}\, B \mid B \notin I\} \models A\}\ , \tag{9}$$

dove i letterali $\texttt{not}\, B$ vengono trattati da $\models$ come se fossero degli atomi qualsiasi.

Si noti che questa è una equazione di punto fisso, poichè I compare in entrambi i lati dell'uguaglianza.

La stessa equazione si può riformulare senza l'insieme $\{\texttt{not}\, B \mid B \notin I\}$, valutando parzialmente P rispetto ai letterali negativi (si ricordi che la valutazione parziale produce programmi equivalenti a quello dato). Durante la valutazione parziale, se un "fatto" $\texttt{not}\, B$ non appartiene a $\{\texttt{not}\, B \mid B \notin I\}$ (cioè se $B \in I$), posso buttare via tutte le regole che hanno $\texttt{not}\, B$ nel corpo, mentre se $\texttt{not}\, B$ appartiene a $\{\texttt{not}\, B \mid B \notin I\}$ (cioè se $B \notin I$), allora posso eliminare $\texttt{not}\, B$ dal corpo della regola (vedasi il capitolo 2.4). Così arriviamo alla seguente definizione.

Definizione 9 La *Gelfond-Lifschitz transformation* o *program reduct* di P rispetto a I, denotata da P^I, è l'insieme delle regole ottenuto da $\mathsf{Ground}(P)$

1. eliminando tutte le regole che nel corpo hanno un letterale $\texttt{not}\, B$ con $B \in I$;
2. eliminando tutti i letterali $\texttt{not}\, B$ dalle regole rimanenti.

Si noti che la Gelfond-Lifschitz transformation, oltre ad effettuare la valutazione parziale di $P \cup \{\texttt{not}\, B \mid B \notin I\}$, rimuove anche tutti i "fatti" negativi da questo programma, il che nel nostro caso è lecito perchè nel lato destro della (9) dobbiamo dimostrare solo letterali positivi, e inoltre tutti i letterali negativi nel corpo delle regole di P sono già stati valutati (quindi, i fatti negativi non servono più).

Il risultato della trasformazione è per definizione un programma positivo; dunque P^I possiede un unico modello minimo, che coincide con l'insieme delle conseguenze logiche di P^I. Pertanto la (9) risulta equivalente all'equazione (10) contenuta nella seguente definizione:

Definizione 10 I è un *modello stabile* di un programa normale P sse

$$I = \mathsf{lm}(P^I)\,. \tag{10}$$

Quando P possiede un unico modello stabile, chiamiamolo M, possiamo definire la relazione $\models_n$ di conseguenza logica nonmonotona che volevamo come segue:

$$P \models_n A \text{ sse } A \in M \text{ e } P \models_n \texttt{not}\, A \text{ sse } A \notin M\,.$$

La proprietà desiderata (7) è soddisfatta per definizione. Per vedere che anche la proprietà desiderata (8) è soddisfatta, si noti che quando P è positivo, allora per ogni I, $P^I = P$. Ne segue che l'unico modello stabile di un P positivo è $\mathsf{lm}(P)$, cioè la sua semantica classica.

Purtroppo, in generale, un programma normale P può non avere modelli stabili, oppure può averne più di uno. In questo caso non è chiaro quale può essere l'insieme di conseguenze logiche nonmonotone del programma.

Esercizio 3 *Verificare che il programma* $P = \{p \leftarrow \texttt{not}\, p\}$ *non ha modelli stabili.*

Esercizio 4 *Verificare che il programma* $P = \{p \leftarrow q, \texttt{not}\, p\}$ *ha esattamente un modello stabile.*

Esercizio 5 *Verificare che il programma* $P = \{p \leftarrow \texttt{not}\, q,\ q \leftarrow \texttt{not}\, p\}$ *ha due modelli stabili.*

Nell'ambito dell'Intelligenza Artificiale, si sono trovate semantiche appropriate per i programmi che hanno un numero di modelli stabili diverso da uno. Sono semantiche che in generale non soddisfano la proprietà desiderabile (7). Nel campo della sicurezza, invece, si preferiscono classi di programmi ristrette

che godono della proprietà (7) e quindi garantiscono che per ogni autorizzazione codificata da un atomo A, (i) sia chiaro se A è derivabile o no dalla politica, e (ii) la non derivabilità sia controllabile nel corpo delle regole mediante un letterale $\texttt{not}\, A$.

Un altro potenziale problema della semantica dei modelli stabili risiede nella sua complessità computazionale intrinseca. Decidere se P possegga almeno un modello stabile, e decidere se ne esista uno che contiene un atomo dato A, sono entrambi problemi NP-completi, mentre decidere se un A dato appartiene a tutti i modelli stabili di P è un problema coNP-completo.

Per ovviare a tutte queste difficoltà ci si limita spesso all'uso di una classe ristretta di programmi, detti *stratificati*, che possiede sempre esattamente un modello stabile, calcolabile in tempo polinomiale. A questi programmi è dedicata la prossima sezione.

3.1 Programmi stratificati

Un programma P è *localmente stratificato* se possiede un *atomic level mapping*, ossia una funzione λ dagli atomi ground in $\{0, \ldots, n\}$ (per qualche $n \geq 0$) che soddisfa le seguenti proprietà, per ogni coppia di atomi A e B:

1. se esiste una regola $(A \leftarrow \ldots, B, \ldots) \in \mathsf{Ground}(P)$, allora $\lambda(A) \geq \lambda(B)$;
2. se esiste $(A \leftarrow \ldots, \texttt{not}\, B, \ldots) \in \mathsf{Ground}(P)$, allora $\lambda(A) > \lambda(B)$.

Il level mapping induce una partizione di $\mathsf{Ground}(P)$ in $P_0 \cup \ldots \cup P_n$, dove P_i è l'insieme delle regole $R \in \mathsf{Ground}(P)$ la cui testa A soddisfa $\lambda(A) = i$ $(0 \leq i \leq n)$.

I programmi localmente stratificati posseggono sempre esattamente un modello stabile. La stratificazione permette di calcolarlo per livelli, cominciando dal livello zero. Intuitivamente, grazie alla stratificazione, le conseguenze di ogni livello dipendono solo dagli atomi definiti nei livelli precedenti (diciamo che un atomo è definito in un programma se compare nella testa di qualche regola di quel programma).

Esercizio 6 *Mostrare che dalla definizione di atomic level mapping segue che P_0 deve essere un programma positivo.*

Per calcolare il modello stabile, definiamo

$$\begin{aligned} I_0 &= \mathsf{lm}(P_0) \\ I_i &= \mathsf{lm}(I_{i-1} \cup P_i^{I_{i-1}}) \quad (1 \leq i \leq n)\,. \end{aligned}$$

Si può dimostrare che I_n è l'unico modello stabile di P.

Esercizio 7 *Si consideri il programma*

$$P = \{p \leftarrow \texttt{not}\, q,\ r \leftarrow \texttt{not}\, p,\ s \leftarrow p, \texttt{not}\, q\}\,.$$

Verificare che P è stratificato dalla funzione λ tale che

$$\lambda(q) = 0,\ \lambda(p) = 1,\ \lambda(r) = \lambda(s) = 2\,.$$

e calcolare con il metodo sopra descritto il modello stabile di P.

Per valutare la complessità dell'algoritmo per il calcolo dei modelli stabili dei programmi stratificati, possiamo assumere senza perdita di generalità che il massimo livello n sia al più uguale al numero di atomi ground (in pratica, "ottimizziamo" λ in modo che sia surgettiva e che quindi i programmi P_i siano tutti non vuoti). Dunque il numero di livelli è dominato dal numero di atomi ground che a sua volta—se il programma è ground—è dominato dalla dimensione del programma. Ora, poichè il calcolo del modello minimo di un programma ground di dimensione m è $O(m \log m)$, e poichè il numero di livelli è dominato da m, abbiamo che la sequenza degli I_i può essere calcolata in tempo $O(m^2 \log m)$.

Resta da vedere quanto è complesso capire se un programma è stratificato.

L'esistenza di un atomic level mapping può essere decisa esaminando un grafo etichettato, detto *grafo delle dipendenze*, i cui nodi sono gli atomi che occorrono in $\mathsf{Ground}(P)$; il grafo delle dipendenze possiede:

- un arco da B ad A etichettato con $+$ (detto arco positivo) se esiste una regola $(A \leftarrow \ldots, B, \ldots) \in \mathsf{Ground}(P)$;
- un arco da B ad A etichettato con $-$ (detto arco negativo) se esiste una regola $(A \leftarrow \ldots, \mathtt{not}\, B, \ldots) \in \mathsf{Ground}(P)$.

Un programma P è *localmente stratificabile* se il suo grafo delle dipendenze non contiene cicli con almeno un arco negativo. Si può dimostrare che per ogni programma normale P, P è localmente stratificato sse P è localmente stratificabile.

Questa proprietà è importante perchè consente di decidere in tempo polinomiale—tramite semplici adattamenti di algoritmi classici per il calcolo delle componenti connesse dei grafi—se un programma dato è stratificato (il programma è stratificabile sse nessuna componente connessa contiene archi negativi).

Esercizio 8 *Si verifichi che i programmi negli esercizi 3 e 5 non sono localmente stratificabili.*

Se P è localmente stratificabile, allora un atomic level mapping λ che stratifica P può essere costruito come segue.

1. si calcolino le componenti connesse del grafo delle dipendenze (cioè gli insiemi di nodi reciprocamente raggiungibili attraverso cammini orientati);
2. si selezionino le componenti connesse che non hanno archi entranti, e per ogni atomo A in tali componenti si ponga $\lambda(A) := 0$;
3. si ponga *livello* $:= 1$;

4. si selezionino le componenti connesse i cui archi entranti provengono solo dalle componenti selezionate ai passi precedenti;

   per ogni atomo A nelle componenti qui selezionate si ponga $\lambda(A) := \mathit{livello}$;

5. se rimangono componenti non selezionate, si ponga $\mathit{livello} := \mathit{livello} + 1$ e si torni al passo 4; altrimenti l'algoritmo termina.

Esercizio 9 *Si applichi questo algoritmo al programma dell'esercizio 7, e si confronti l'atomic level mapping prodotto con quello proposto nell'esercizio 7.*

In generale, un dato programma P può essere stratificato da più di un atomic level mapping λ.

Esercizio 10 *Si trovi un diverso atomic level mapping per il programma dell'esercizio 7.*

3.2 Programmi normali e politiche di sicurezza

Riferimenti bibliografici

[1] Michel Minoux. LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters* 29(1): 1-12 (1988)