

14.2 Sintassi

I programmi logici sono insiemi di formule logiche di una particolare forma. Inizieremo quindi con alcune nozioni di base necessarie per definirne la sintassi.

La logica della quale ci occupiamo è quella del prim'ordine, detta anche calcolo dei predicati. La terminologia si riferisce al fatto che si usano dei simboli per esprimere (o, come si dice con una terminologia un poco obsoleta “predicare”) proprietà di elementi che fanno parte di un dominio del discorso \mathcal{D} prefissato. Logiche più espansive (del secondo ordine, del terzo ordine ecc.) permettono anche di avere predicati che come argomenti hanno, invece che elementi di \mathcal{D} , oggetti più complicati quali insiemi e funzioni su \mathcal{D} (second'ordine), insiemi di funzioni (terz'ordine) ecc.

14.2.1 Il linguaggio della logica del prim'ordine

Innanzitutto, come per ogni altro sistema formale, per poter parlare del calcolo dei predicati (e quindi dei programmi logici) dobbiamo definirne il linguaggio. Un *linguaggio del prim'ordine* consiste di tre componenti:

1. un *alfabeto*;
2. i *termini* definiti su tale alfabeto;
3. le *formule ben formate* definite su tale alfabeto.

Vediamo in ordine i vari componenti di questa definizione.

Alfabeto L'alfabeto, al solito, è un insieme⁵ di simboli. In questo caso consideriamo tale insieme partizionato in due sottoinsiemi disgiunti: l'insieme dei *simboli logici*, comuni a tutti i linguaggi del prim'ordine, e l'insieme dei *simboli non logici*, specifici di un qualche dominio di interesse. Ad esempio, tutti i linguaggi del prim'ordine, verosimilmente, useranno un simbolo (logico) per indicare la congiunzione; se però stiamo considerando degli ordinamenti su di un insieme, probabilmente fra i simboli non logici del linguaggio vorremo avere anche il simbolo $<$.

L'insieme dei *simboli logici* contiene i seguenti elementi

- i connettivi logici \wedge (congiunzione), \vee (disgiunzione), \neg (negazione), \rightarrow (implicazione) e \leftrightarrow (doppia implicazione);
- le costanti proposizionali *true* e *false*;
- i quantificatori \exists (esiste) e \forall (per ogni);
- alcuni simboli di interruzione quali le parentesi $(,)$ e la virgola $,$;
- un insieme infinito (numerabile) V di *variabili*, indicate con X, Y, Z, \dots

⁵Tutti gli insiemi che considereremo nel seguito sono finiti o numerabili.

I simboli *non logici* sono definiti da una *segnatura con predicati* $\langle \Sigma, \Pi \rangle$: si tratta di una coppia nella quale il primo elemento Σ è la *segnatura delle funzioni*, cioè un insieme di simboli di funzione, considerati ognuno con la propria arietà⁶. Il secondo elemento della coppia Π è la *segnatura dei predicati*, un insieme di simboli di predicato insieme alle loro arietà. Le funzioni di arietà 0 sono dette *costanti* e sono indicate con le lettere a, b, c, \dots . I simboli di funzione di arietà positiva sono di solito indicati da f, g, h, \dots , mentre i simboli di predicato sono indicati da p, q, r, \dots . Assumiamo che gli insiemi Σ e Π abbiano intersezione vuota e siano anche disgiunti dagli altri insiemi di simboli elencati in precedenza. La differenza fra simboli di funzione e di predicato è che i primi devono essere interpretati come funzioni, mentre i secondi devono essere interpretati come relazioni. Questa distinzione sarà più chiara quando parleremo di formule.

Termini La nozione di termine, fondamentale nella logica matematica, nell'ambito dell'informatica è usata implicitamente in molti contesti. Ad esempio, un'espressione aritmetica in sostanza è un termine ottenuto applicando degli operatori (aritmetici) a degli operandi. Anche altri tipi di costrutti, quali le stringhe, gli alberi binari, le liste ecc. possono essere convenientemente viste come termini, ottenuti a partire da opportuni costruttori.

Nel caso più semplice un termine è ottenuto applicando un simbolo di funzione a variabili e costanti in modo tale da rispettare l'arietà. Ad esempio, se a e b sono costanti, X e Y sono variabili e f e g hanno arietà 2, $f(a, b)$ e $g(a, X)$ sono termini. Nulla però vieta di usare termini come argomenti di una funzione, purché sia sempre rispettata l'arietà. Possiamo ad esempio scrivere $g(f(a, b), Y)$ o anche $g(f(a, f(X, Y)), X)$ e così via.

Nel caso più generale possiamo definire i termini come segue.

Definizione 14.1 (Termini) *I termini sulla segnatura Σ (e sull'insieme di variabili V) sono definiti induttivamente⁷ come segue:*

- *una variabile (in V) è un termine;*
- *se f (in Σ) è un simbolo di funzione di arietà n e t_1, \dots, t_n sono termini, allora $f(t_1, \dots, t_n)$ è un termine.*

Come caso particolare del secondo punto, una costante è un termine: stando alla lettera, un termine che corrisponde ad una costante si dovrebbe scrivere con le parentesi: $a(), b(), \dots$; stabiliamo, per convenzione, che nel caso di simboli di funzione di arietà 0 le parentesi sono omesse. I termini senza variabili sono detti termini *ground*. I termini usualmente sono denotati dalle lettere s, t, u, \dots . Si noti che nei termini non compaiono predicati: questi compariranno nelle formule (per esprimere proprietà dei termini).

⁶Ricordiamo che, come già detto, l'arietà indica il numero di argomenti di una funzione o di una relazione.

⁷Si veda il riquadro di pag. 233 per le definizioni induttive.

Formule Le *formule ben formate* (*formule* in breve) del linguaggio ci permettono di esprimere proprietà dei termini che poi, dal punto di vista semantico, sono proprietà di un particolare dominio di interesse. Ad esempio, se abbiamo il predicato $>$, interpretato nel modo usuale, scrivendo $>(3, 2)$ vogliamo esprimere il fatto che il termine 3 corrisponde ad un valore che è maggiore di quello associato al termine 2. I predicati possono poi essere usati per costruire espressioni complesse mediante i simboli logici. Ad esempio, la formula $>(X, Y) \wedge >(Y, Z) \rightarrow >(X, Z)$ esprime la transitività di $>$, in quanto afferma che se vale $>(X, Y)$ e vale anche $>(Y, Z)$, allora vale $>(X, Z)$.

Volendo definire con precisione le formule, abbiamo innanzitutto le formule atomiche (o atomi), costruite applicando un predicato a termini in modo tale da rispettare l'arità. Ad esempio, se p ha arità 2, usando due termini introdotti prima possiamo scrivere $p(f(a, b), f(a, X))$. Usando i connettivi logici ed i quantificatori, possiamo costruire formule complesse a partire dalle formule atomiche. Al solito, possiamo usare una definizione induttiva (oppure, equivalentemente, una grammatica libera, si veda l'Esercizio 1).

Definizione 14.2 (Formule) Le formule (*ben formate*) del linguaggio sulla segnatura con termini (Σ, Π) sono definite induttivamente come segue:

1. se t_1, \dots, t_n sono termini sulla segnatura Σ e $p \in \Pi$ è un simbolo di predicato di arità n , allora $p(t_1, \dots, t_n)$ è una formula;
2. true e false sono formule;
3. se F e G sono formule allora $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ e $(F \leftrightarrow G)$ sono formule;
4. se F è una formula e X è una variabile, allora $\forall X.F$ e $\exists X.F$ sono formule.

14.2.2 I programmi logici

Una formula della logica del prim'ordine può avere una struttura molto complessa, che influisce sulla difficoltà di determinare una dimostrazione per essa. Nell'ambito della dimostrazione automatica di teoremi e, poi, nel contesto della programmazione logica, sono state identificate particolari classi di formule, dette *clausole*, che si prestano ad essere manipolate più efficientemente, in specie usando una particolare regola di dimostrazione detta *risoluzione*. A noi interessano, in particolare, delle versioni ristrette della nozione di clausola (le *clausole definite*) e della regola di risoluzione (la *risoluzione SLD*, si veda l'Approfondimento 14.1) per le quali il procedimento di ricerca di una dimostrazione non solo è particolarmente semplice, ma permette di calcolare esplicitamente i valori delle variabili necessari alla dimostrazione. Questi valori possono essere considerati come il risultato della computazione, dando luogo ad un modello di calcolo interamente basato sulla deduzione logica. Vedremo meglio tale modello più avanti, per ora ci concentriamo sugli aspetti sintattici.

Definizione 14.3 (Programma logico) Siano H, A_1, \dots, A_n formule atomiche. Una clausola definita (per noi semplicemente "clausola") è una formula della

$$H : -A_1, \dots, A_n.$$

forma

Se $n = 0$ la clausola è detta unitaria, o fatto, ed il simbolo $: -$ è omesso (ma non il punto terminale). Un programma logico è un insieme di clausole, mentre un programma PROLOG puro è una sequenza di clausole. Una query (o goal) è una sequenza di atomi A_1, \dots, A_n .

Chiariamo alcuni punti nella precedente definizione. Innanzitutto il simbolo $: -$, che non avevamo introdotto nel nostro alfabeto, è semplicemente un sostituto per l'implicazione (rovesciata) \leftarrow ed è quello comunemente usato nei linguaggi logici reali⁸.

La virgola in una clausola o in una query dal punto di vista logico è da intendersi come congiunzione. La notazione " $H : -A_1, \dots, A_n.$ " è dunque una abbreviazione per " $H \leftarrow A_1 \wedge \dots \wedge A_n.$ ". Si noti che il punto fa parte della notazione ed è importante, in quanto segnala all'eventuale interprete o compilatore quando termina la clausola che si sta considerando.

La parte a sinistra di $: -$ è detta *testa* della clausola mentre quella a destra è detta *corpo*. Un fatto è dunque una clausola con corpo vuoto. Un programma è un insieme di clausole, nel caso del formalismo teorico. Nel caso di PROLOG invece un programma è considerato come una sequenza perché, come vedremo, in questo caso l'ordine delle clausole ha rilevanza. Qui abbiamo usato la terminologia semplificata usata in molti testi recenti (ad esempio [9]). Per la terminologia più precisa si veda il riquadro successivo. L'insieme delle clausole che contengono nella testa il simbolo di predicato p è detto la *definizione* di p . Le variabili che occorrono nel corpo di una clausola e non nella testa sono dette variabili *locali*.

14.3 Teoria dell'unificazione

Il meccanismo fondamentale di computazione nella programmazione logica è la soluzione di equazioni fra termini, realizzata mediante il processo di unificazione. In tale processo vengono calcolate delle sostituzioni che permettono di legare (o istanziare) delle variabili a dei termini. La composizione delle varie sostituzioni ottenute nel corso della computazione fornisce il risultato del calcolo. Prima di vedere nel dettaglio il modello computazionale del paradigma logico dobbiamo quindi analizzare alcuni aspetti riguardanti l'unificazione, cosa che facciamo in questo paragrafo.

14.3.1 La variabile logica

Prima di entrare nel dettaglio del processo di unificazione, è importante chiarire che la nozione di variabile che qui stiamo considerando è diversa da quella vista

⁸Si usa $: -$ per motivi pragmatici: sulle tastiere, soprattutto al tempo in cui vennero introdotti i linguaggi logici, era molto più semplice realizzare un $: -$ che una freccia a sinistra.

Clausole

Il lettore familiare con la logica del prim'ordine avrà riconosciuto nella nozione di clausola che abbiamo dato (Definizione 14.3) un caso particolare di quella che si usa in logica. Un clausola, in senso generale, è infatti una formula del tipo

$$\forall X_1, \dots, X_m (L_1 \vee L_2 \vee \dots \vee L_n)$$

dove L_1, \dots, L_n sono *letterali*, ovvero atomi oppure atomi negati, e X_1, \dots, X_n sono tutte le variabili che occorrono in L_1, \dots, L_n . Per maggiore chiarezza, se pariamo gli atomi negati dagli altri e dunque vediamo una clausola come una formula della forma

$$\forall X_1, \dots, X_m (A_1 \vee A_2 \vee \dots \vee A_m, \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_k).$$

Usando la nota equivalenza della logica che permette di esprimere l'implicazione in termini di una disgiunzione, possiamo esprimere una tale formula nella seguente forma speciale, equivalente alla precedente

$$A_1, \dots, A_m \leftarrow B_1, \dots, B_k. \quad (14.1)$$

Una *clausola di programma*, detta anche *clausola definita*, è una clausola che ha un solo atomo non negato; nella forma espressa dalla (14.1) una clausola definita ha sempre $m = 1$, che è proprio la nozione da noi introdotta con la Definizione 14.3. Un fatto è dunque un clausola definita senza atomi negativi. Infine una *clausola negativa*, detta anche *query o goal*, è una clausola in forma (14.1) nella quale $m = 0$.

nel Paragrafo 8.2.1. Qui consideriamo la cosiddetta *variabile logica* che, analogamente alla variabile di cui abbiamo già parlato, costituisce un'incognita che può assumere i valori di un insieme prefissato. Nel nostro caso tale insieme è quello dei termini definiti sull'alfabeto dato. Questo, unitamente all'uso che della variabile logica viene fatto nel paradigma logico, fa sì che vi siano tre importanti differenze fra questa nozione e la variabile modificabile dei linguaggi imperativi:

1. la variabile logica può essere legata una sola volta, nel senso che se una variabile è legata ad un termine questo legame non può essere distrutto (può essere eventualmente modificato il termine, come discutiamo più avanti). Ad esempio, se in un programma logico leghiamo la variabile X alla costante a , tale legame non può poi essere sostituito da un altro che lega X alla costante b . Questo invece, ovviamente, è possibile nei linguaggi imperativi mediante il comando di assegnamento. Il fatto che il legame di una variabile non possa essere eliminato non significa però che non si possa modificare il valore della variabile. Questa apparente contraddizione merita di essere considerata con qualche attenzione, come facciamo nel punto seguente.

2. Il valore di una variabile logica può essere definito parzialmente (o indefinito) per essere poi ulteriormente specificato in seguito. Questo perché un termine legato ad una variabile può contenere, a sua volta, altre variabili logiche. Ad esempio, se la variabile X è legata al termine $f(Y, Z)$, successivi legami delle variabili Y e Z modificano anche il valore della variabile X (se Y viene legata ad a e Z viene legata a $g(W)$, il valore di X diverrà il termine $f(a, g(W))$), e il discorso potrebbe continuare modificando il valore della W). Questo meccanismo di specifica del valore di una variabile per approssimazioni successive, per così dire, è tipico dei linguaggi logici ed è molto diverso da quello che si ha nel contesto imperativo, dove un valore assegnato ad una variabile non può essere parzialmente definito.
3. Una terza importante differenza riguarda la natura bidirezionale dei legami nel caso delle variabili logiche. Se X è legata al termine $f(Y)$ e successivamente proviamo a legare X al termine $f(a)$, l'effetto che produciamo è quello di legare la variabile Y alla costante a . Questo non contraddice il primo punto, dato che il legame di X al termine $f(Y)$ non viene distrutto, ma viene solo ulteriormente specificato il valore $f(Y)$ per mezzo del legame effettuato per la Y . Quindi, non solo possiamo modificare il valore di una variabile modificando il termine ad essa legato, ma possiamo anche modificare tale termine fornendo un altro legame per la variabile stessa; ovviamente tale secondo legame deve essere consistente con il primo, cioè se X è legata al termine $f(Y)$ non possiamo cercare di legare X ad un termine della forma $g(Z)$.

L'ultimo punto, fondamentale, è quello che permette di usare in molti modi diversi, come vedremo, uno stesso programma logico. In sostanza si tratta di una caratteristica dovuta alla presenza del meccanismo di unificazione, del quale parleremo fra un attimo. Prima però dobbiamo introdurre la nozione di sostituzione.

14.3.2 Sostituzione

Il legame fra variabili e termini è realizzato mediante la nozione di sostituzione, che, come dice il nome stesso, permette di "sostituire" un termine a 1 posto di una variabile. Una sostituzione, indicata usualmente con le lettere greche $\vartheta, \sigma, \rho, \dots$ può essere definita come segue.

Definizione 14.4 (Sostituzione) *Un sostituzione è una funzione da variabili a termini tale che il numero di variabili che non sono mappate in se stesse è finito. Indichiamo una sostituzione ϑ usando la notazione*

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

dove X_1, \dots, X_n sono variabili diverse, t_1, \dots, t_n sono termini e dove assumiamo che t_i sia diverso da X_i , per $i = 1, \dots, n$.

Nella precedente definizione una coppia X_i/t_i è detta legame (binding); nel caso in cui tutti i t_1, \dots, t_n siano termini ground allora ϑ è detta sostituzione ground;

ε denota la sostituzione vuota. Per ϑ rappresentata come nella Definizione 14.4, definiamo il dominio, il codominio e le variabili della sostituzione come segue:

$$\begin{aligned} \text{Dominio}(\vartheta) &= \{X_1, \dots, X_n\}, \\ \text{Codominio}(\vartheta) &= \{Y \mid Y \text{ è una variabile in } t_i, \text{ per qualche } t_i, 1 \leq i \leq n\}, \end{aligned}$$

Una sostituzione può essere applicata ad un termine o, più in generale, ad una qualsiasi espressione sintattica, per modificare il valore delle variabili presenti nel dominio della sostituzione. Più precisamente, se consideriamo una generica espressione E (che, può essere un termine, un letterale, una congiunzione di atomi ecc.) il risultato dell'applicazione di $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ ad E , indicato con $E\vartheta^9$, è ottenuto rimpiazzando simultaneamente tutte le occorrenze di X_i in E mediante il corrispondente t_i , per ogni $1 \leq i \leq n$. Così, ad esempio, se applichiamo la sostituzione $\vartheta = \{X/a, Y/f(W)\}$ al termine $g(X, W, Y)$ otteniamo il termine $g(X, W, Y)\vartheta$ e cioè $g(a, W, f(W))$. Si noti che l'applicazione è simultanea: ad esempio, se applichiamo la sostituzione $\sigma = \{Y/f(X), X/a\}$ al termine $g(X, Y)$ otteniamo $g(a, f(X))$ (e non $g(a, f(a))$).

La *composizione* $\vartheta\sigma$ di due sostituzioni $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ e $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$ è definita come la sostituzione ottenuta rimuovendo dall'insieme

$$\{X_1/t_1\sigma, \dots, X_n/t_n\sigma, Y_1/s_1, \dots, Y_m/s_m\}$$

le coppie $X_i/t_i\sigma$ tali che X_i è eguale a $t_i\sigma$ e le coppie Y_i/s_i tali che $Y_i \in \{X_1, \dots, X_n\}$. La composizione è associativa e non è difficile rendersi conto che, per una qualsiasi espressione E , vale che $E(\vartheta\sigma) = (E\vartheta)\sigma$: l'effetto dell'applicazione della composizione è lo stesso che si ottiene applicando successivamente le due sostituzioni che vogliamo comporre.

Ad esempio, componendo

$$\vartheta_1 = \{X/f(Y), W/a, Z/X\} \text{ e } \vartheta_2 = \{Y/b, W/b, X/Z\}$$

otteniamo la sostituzione

$$\vartheta = \vartheta_1\vartheta_2 = \{X/f(b), W/a, Y/b\}.$$

Se applichiamo quest'ultima sostituzione al termine $g(X, Y, W)$ otteniamo il termine $g(f(b), b, a)$, così come lo stesso termine si ottiene applicando a $g(X, Y, W)$ prima ϑ_1 e poi ϑ_2 . Si noti che, nel risultato ϑ della composizione, la Y presente in ϑ_1 viene istanziata a b per effetto del legame presente in ϑ_2 ; la X presente in Z/X viene istanziata a Z per effetto del legame X/Z presente in ϑ_2 , dopo di che

⁹Si noti che sia per quanto riguarda la notazione usata per le sostituzioni sia per quella relativa all'applicazione di una sostituzione, vi sono versioni opposte: la notazione per la sostituzione di un termine N al posto della variabile X usata a pagina 441 è N/X , in accordo allo standard del paradigma funzionale. Qui invece tale legame è indicato con X/N . In generale, qui usiamo la notazione più comune nell'ambito della programmazione logica.

il legame Z/Z viene eliminato dalla sostituzione risultante (perché realizza un'identità); i legami W/b e X/Z presenti in ϑ_2 infine scompaiono da ϑ' perché sia W che Z compaiono nel dominio di (ossia, a sinistra di un legame in) ϑ_1 .

Un particolare tipo di sostituzioni è costituito da quelle che semplicemente ridenominano le variabili. Ad esempio, la sostituzione $\{X/W, W/X\}$ altro non fa che cambiar di nome alle variabili X e W . Tali sostituzioni sono dette ridenominazioni (o renaming) e possono essere definite come segue.

Definizione 14.5 (Ridenominazione) Una sostituzione ρ è una ridenominazione se esiste la sua sostituzione inversa ρ^{-1} tale che $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$.

Si noti che la sostituzione $\{X/Y, W/Y\}$ non è un renaming: essa infatti non solo cambia nome a due variabili, ma fa diventare eguali due variabili che in precedenza erano diverse.

Infine ci sarà utile definire un preordine \leq sulle sostituzioni dove $\vartheta \leq \sigma$ è letto come ϑ è più generale di σ . Definiamo dunque $\vartheta \leq \sigma$ se (e solo se) esiste una sostituzione γ tale che $\vartheta\gamma = \sigma$. Analogamente, date due espressioni t e t' , definiamo $t \leq t'$ (t è più generale di t') se e solo se esiste ϑ tale che $t\vartheta = t'$. La relazione \leq è un preordine e l'equivalenza da esso indotta¹⁰ è detta varianza: t e t' sono dunque varianti se t è un'istanza di t' e, viceversa, t' è un'istanza di t . Non è difficile vedere che questa definizione è equivalente a dire che t e t' sono varianti se esiste un renaming ρ tale che t è sintatticamente identico a $t'\rho$. Queste definizioni possono essere estese a generiche espressioni in modo ovvio. Infine, se ϑ è una sostituzione che ha come dominio l'insieme di variabili V e W è un sottoinsieme di V , la *restrizione* di ϑ alle variabili in W è la sostituzione ottenuta considerando solo i legami per le variabili in W , ossia è la sostituzione definita come segue

$$\{Y/t \mid Y \in W \text{ e } Y/t \in \vartheta\}.$$

Un confronto con il paradigma imperativo può aiutare a comprendere meglio le nozioni che stiamo qui considerando. Come abbiamo visto nel Paragrafo 2.5, nel paradigma imperativo la semantica può essere espressa facendo riferimento ad una nozione di stato che associa ad ogni variabile un valore¹¹. Un'espressione contente delle variabili è valutata rispetto ad uno stato per ottenere un valore completamente definito.

Nel paradigma logico l'associazione dei valori alle variabili è realizzata mediante le sostituzioni. L'applicazione di una sostituzione ad un termine (o ad una espressione più complessa) può essere vista come la valutazione del termine, valutazione che restituisce un altro termine e quindi, in generale, un valore parzialmente definito.

¹⁰Dato un preordine \leq , la relazione di equivalenza *indotta da* \leq è definita come $t \equiv t'$ se (e solo se) $t \leq t'$ e $t' \leq t$.

¹¹A voler essere precisi, come abbiamo visto nel riquadro di pag. 214, nei linguaggi reali tale associazione è realizzata mediante due funzioni, ambiente e memoria. Questo comunque non cambia la sostanza di quello che stiamo dicendo.

14.3.3 L'unificatore più generale

Il meccanismo di computazione di base del paradigma logico è la valutazione di equazioni della forma $s = t^{12}$, dove s e t sono termini e $=$ è un simbolo di predicato interpretato come uguaglianza sintattica sull'insieme di tutti i termini ground, insieme detto anche *universo di Herbrand*¹³. Vediamo di chiarire meglio il senso di queste uguaglianze.

Se in un programma logico scriviamo $X = a$ intendiamo dire che la variabile X deve essere legata alla costante a . La sostituzione $\{X/a\}$ costituisce dunque una soluzione per tale equazione dato che, applicando questa sostituzione all'equazione, otteniamo $a = a$ che è un'equazione evidentemente soddisfatta. L'analogia sintattica con l'assegnamento dei linguaggi imperativi non deve trarre in inganno, dato che si tratta di una nozione completamente diversa. Infatti, a differenza di quello che possiamo fare in un linguaggio imperativo, qui possiamo anche scrivere $a = X$ invece di $X = a$, ed il significato non cambia (l'uguaglianza che consideriamo è simmetrica, come tutte le relazioni di uguaglianza). Anche l'analogia con l'uguaglianza delle espressioni aritmetiche può essere, per certi versi, fuorviante, come illustrato dal seguente esempio. Supponiamo di avere un simbolo di funzione binario $+$ che, intuitivamente, esprime la somma di due numeri naturali e consideriamo l'equazione $3 = 2 + 1$, dove, per comodità, usiamo la notazione infissa per il $+$ e rappresentiamo nel modo usuale i numeri naturali. Dato che l'equazione $3 = 2 + 1$ non contiene variabili, essa può essere o vera (ossia, già risolta) o falsa (cioè non risolvibile). Contrariamente a quello che l'intuizione aritmetica ci suggerirebbe, in un programma logico (puro) tale equazione non è risolvibile. Questo perché, come abbiamo detto, il simbolo $=$ è interpretato come uguaglianza sintattica sull'insieme dei termini ground (universo di Herbrand): è evidente che, dal punto di vista sintattico, la costante 3 è diversa dal termine $2 + 1$ e trattandosi di termini ground (cioè completamente istanziati) non c'è alcun modo di renderli sintatticamente eguali. Analogamente, l'equazione $f(X) = g(Y)$ non ha soluzioni (non è risolvibile) perché comunque si istanzino le variabili X e Y non possiamo rendere eguali i due diversi simboli di funzione f e g . Si noti che anche l'equazione $f(X) = f(g(X))$ non ha soluzioni, perché la variabile X nel termine di sinistra dovrebbe essere istanziata con $g(X)$ e quindi l'eventuale soluzione dovrebbe contenere la sostituzione $\{X/g(X)\}$; però l'applicazione di tale sostituzione al termine di destra istanzierebbe la X , producendo il termine $f(g(g(X)))$, dunque la X nel termine di sinistra dovrebbe essere istanziata a $g(g(X))$ invece che a $g(X)$ e così via, senza mai giungere ad una

¹² Si faccia attenzione alla notazione che sovraccarica, come usuale, il simbolo “ $=$ ”: scrivendo $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$ intendiamo dire che ϑ è la sostituzione $\{X_1/t_1, \dots, X_n/t_n\}$, mentre scrivendo $s = t$ indichiamo un'equazione.

¹³ Il simbolo $=$ usualmente usa la notazione infissa per migliorare la leggibilità. I vari linguaggi logici reali possono avere delle leggere differenze sintattiche al riguardo e usare anche vari simboli di uguaglianza, con significati diversi. Qui facciamo riferimento alla programmazione logica “pura”.

soluzione¹⁴. In generale dunque l'equazione $X = t$ non è risolvibile se t contiene la variabile X (ed è diverso da X).

Se invece consideriamo l'equazione $f(X) = f(g(Y))$ questa è risolvibile: una soluzione è la sostituzione $\vartheta = \{X/g(Y)\}$ perché se questa è applicata ai due termini dell'equazione li rende sintatticamente identici: infatti $f(X)\vartheta$ è identico a $f(g(Y))$ che è identico $f(g(Y))\vartheta$. Detto in termini più formali, la sostituzione ϑ unifica i due termini dell'equazione ed è pertanto detta *unificatore*. Si noti che abbiamo detto "una soluzione" perché vi sono molte (infinite) sostituzioni che sono unificatori di X e $g(Y)$: basta istanziare Y nella definizione di ϑ . Così, ad esempio, sono unificatori anche le sostituzioni $\{X/g(a), Y/a\}$, $\{X/g(f(Z)), Y/f(Z)\}$, $\{X/g(f(a)), Y/f(a)\}$ e così via. Tutte queste sostituzioni sono però *meno generali* di ϑ secondo il preordine che abbiamo definito in precedenza: ognuna di esse, cioè, si può ottenere componendo ϑ con un'opportuna sostituzione. Ad esempio, $\{X/g(a), Y/a\}$ è eguale a $\vartheta\{Y/a\}$ (indichiamo la composizione di sostituzioni con la giustapposizione, come già visto). In questo senso, diciamo che ϑ è l'unificatore più generale, o l'm.g.u., (*most general unifier*) di X e $g(Y)$.

Prima di passare alla definizioni generali, si noti un ultimo particolare importante: il processo di soluzione di un'equazione, e cioè di unificazione, può creare dei legami di natura bidirezionale, ossia non è specificata una direzione nella quale si devono realizzare le associazioni. Ad esempio, una soluzione dell'equazione $f(X, a) = f(b, Y)$ è data dalla sostituzione $\{X/b, Y/a\}$, dove si lega una variabile a sinistra ed una a destra del simbolo $=$. Invece nell'equazione $f(X, a) = f(Y, a)$ possiamo scegliere se legare la variabile a sinistra (usando $\{X/Y\}$) oppure quella a destra (usando $\{Y/X\}$) o anche tutte e due (usando $\{X/Z, Y/Z\}$) nella soluzione dell'equazione.

Quest'aspetto, sul quale ritorneremo, è importante perché permette di realizzare un meccanismo di passaggio dei parametri bidirezionale e, caratteristica unica del paradigma logico, di usare uno stesso programma in molti modi diversi, facendo diventare gli argomenti di input argomenti di output e vice versa, senza alcuna modifica al programma stesso.

Vediamo ora le definizioni formali.

Definizione 14.6 (M.g.u.) Dato un insieme di equazioni $E = \{s_1 = t_1, \dots, s_n = t_n\}$, dove s_1, \dots, s_n e t_1, \dots, t_n sono termini, la sostituzione ϑ è un unificatore per E se le sequenze $(s_1, \dots, s_n)\vartheta$ e $(t_1, \dots, t_n)\vartheta$ sono sintatticamente identiche. Un unificatore di E è detto unificatore più generale (m.g.u.) se è più generale di ogni altro unificatore di E , ossia se per ogni altro unificatore σ di E vale che σ è eguale a $\vartheta\tau$ per un'opportuna sostituzione τ .

La precedente nozione di unificatore può essere estesa ad altri oggetti sintattici nel modo ovvio. In particolare, diremo che ϑ è un unificatore di due atomi $p(s_1, \dots, s_n)$ e $p(t_1, \dots, t_n)$ se ϑ è un unificatore per $\{s_1 = t_1, \dots, s_n = t_n\}$.

¹⁴Si noti che ammettendo termini infiniti potremmo invece trovare una soluzione.

14.3.4 Un algoritmo di unificazione

Un risultato importante, dovuto a Robinson nel 1965, dimostra che è decidibile il problema di verificare se un insieme di equazioni fra termini è unificabile. La dimostrazione è costruttiva, nel senso che fornisce un *algoritmo di unificazione* che, per ogni insieme di equazioni, produce il loro m.g.u. se l'insieme è unificabile e riporta un fallimento in caso contrario¹⁵.

Si può anche dimostrare che un m.g.u. è unico a meno di renaming. L'algoritmo di unificazione che adesso vediamo non è quello originale di Robinson, ma è quello di Martelli e Montanari, del 1982, che riprende alcune idee presenti già nella tesi di Herbrand del 1930.

Algoritmo di unificazione di Martelli e Montanari Dato un insieme di equazioni

$$E = \{s_1 = t_1, \dots, s_n = t_n\},$$

l'algoritmo produce o un fallimento, o un insieme di equazioni in forma risolta del tipo

$$\{X_1 = r_1, \dots, X_m = r_m\},$$

dove X_1, \dots, X_m sono variabili tutte diverse che non compaiono nei termini r_1, \dots, r_m . L'insieme di equazioni è equivalente all'insieme di partenza E e da esso possiamo ottenere un m.g.u. per E semplicemente interpretando ogni uguaglianza come un legame; dunque l'm.g.u. cercato è la sostituzione

$$\{X_1/r_1, \dots, X_m/r_m\}.$$

L'algoritmo è non deterministico nel senso che quando vi sono più azioni possibili ne viene scelta una in modo casuale, senza alcuna priorità fra le diverse azioni¹⁶. L'algoritmo è descritto dai seguenti passi.

1. Scegli in modo non deterministico un'equazione all'interno dell'insieme E .
2. A seconda del tipo dell'equazione scelta, esegui, se possibile, una delle operazioni specificate come segue, dove a sinistra del simbolo ":" indichiamo il tipo d'equazione e a destra l'azione associata:
 - (i) $f(l_1, \dots, l_k) = f(m_1, \dots, m_k)$: elimina questa equazione dall'insieme E e aggiungi ad E le equazioni $l_1 = m_1, \dots, l_k = m_k$;
 - (ii) $f(l_1, \dots, l_k) = g(m_1, \dots, m_h)$: se f è diverso da g termina con fallimento;
 - (iii) $X = X$: elimina questa equazione dall'insieme E ;

¹⁵L'algoritmo originale di Robinson considera l'unificazione di due soli termini ma questo, ovviamente, non è riduttivo dato che l'unificazione di $\{s_1 = t_1, \dots, s_n = t_n\}$ può essere vista come l'unificazione di $f(s_1, \dots, s_n)$ e $f(t_1, \dots, t_n)$.

¹⁶Ritorneremo sul non determinismo in breve, quando parleremo della semantica operazionale dei programmi logici.

- (iv) $X = t$: se t non contiene la variabile X e tale variabile appare in un'altra equazione dell'insieme E , applica la sostituzione $\{X/t\}$ a tutte le altre equazioni dell'insieme E ;
 - (v) $X = t$: se t contiene la variabile X termina con fallimento;
 - (vi) $t = X$: se t non è una variabile elimina questa equazione dall'insieme E e aggiungi ad E l'equazione $X = t$.
3. se nessuna delle operazioni precedenti è possibile, termina con successo (E contiene la forma risolta); se invece è stata eseguita un'operazione diversa dalla terminazione con fallimento torna a (1).

Si tratta, come si vede, di un algoritmo molto semplice; discutiamo in dettaglio le varie operazioni (i)-(vi).

Il primo caso è quello di due termini che concordano nel simbolo di funzione più esterno. In questo caso per unificare i due termini dovremo procedere all'unificazione degli argomenti, cosa che facciamo rimpiazzando l'equazione originaria con le equazioni ottenute uguagliando gli argomenti nelle stesse posizioni. Si noti che questo caso include anche le equazioni fra costanti del tipo $a = a$ (dove a è un simbolo di funzione di arietà 0) che vengono eliminate senza aggiungere niente.

Il secondo caso produce un fallimento dato che, essendo f e g diversi, come abbiamo visto i due termini non sono unificabili.

L'equazione $X = X$ è eliminata in quanto soddisfatta dalla sostituzione identica che quindi non produce alcun cambiamento alle altre equazioni.

Il quarto caso è quello più interessante. Un'equazione $X = t$ è già in forma risolta (perché, per ipotesi di questo caso, t non contiene X): in altre parole, la sostituzione $\{X/t\}$ è l'm.g.u. di questa equazione. Perché l'effetto di tale m.g.u. sia combinato con quelli prodotti dalle altre equazioni, dobbiamo applicare la sostituzione $\{X/t\}$ a tutte le altre equazioni dell'insieme E .

Nel caso in cui invece t contenga la variabile X , come abbiamo già detto, l'equazione non ha soluzione e quindi l'algoritmo termina con fallimento. È importante notare che questo controllo, detto *occur check* è assente in molte implementazioni di PROLOG per motivi di efficienza. Dunque, molte implementazioni di PROLOG usano un algoritmo di unificazione non corretto!

L'ultimo caso infine serve solo ad ottenere una forma risolta, nella quale le variabili compaiono a sinistra e i termini a destra del simbolo $=$.

È facile convincersi che l'algoritmo termina, dato che la profondità dei termini di partenza è finita. È inoltre possibile dimostrare che l'algoritmo produce un m.g.u., ottenuto dall'interpretazione della forma risolta finale delle equazioni come sostituzione.

Considerando con attenzione l'algoritmo, ci si rende conto che il calcolo dell'unificatore più generale avviene in modo incrementale, risolvendo equazioni sempre più semplici fino ad arrivare alla forma risolta. È possibile esprimere questo processo anche in termini di composizione di sostituzioni, come in effetti avviene nel modello operazionale dei linguaggi logici. Vediamo questo punto con un esempio, che costituisce anche un esempio di applicazione dell'algoritmo d'unificazione appena visto. Per semplicità selezioneremo sempre la prima

equazione a sinistra (il risultato finale, tuttavia, non dipende da tale assunzione: una qualsiasi altra regola di selezione porterebbe allo stesso risultato, a meno di renaming).

Consideriamo l'insieme di equazioni

$$E = \{f(X, b) = f(g(Y), W), h(X, Y) = h(Z, W)\}.$$

Selezionando la prima equazione a sinistra, mediante l'operazione descritta al punto (i) l'insieme E è trasformato in

$$E_1 = \{X = g(Y), b = W, h(X, Y) = h(Z, W)\};$$

usando l'operazione in (iv) otteniamo quindi

$$E_2 = \{X = g(Y), b = W, h(g(Y), Y) = h(Z, W)\};$$

usando (vi) e poi di nuovo (iv) sulla seconda equazione otteniamo infine

$$E_3 = \{X = g(Y), W = b, h(g(Y), Y) = h(Z, b)\},$$

che già contiene la forma risolta della prima equazione dell'insieme E : difatti la sostituzione

$$\vartheta_1 = \{X/g(Y), W/b\}$$

è un m.g.u. per $f(X, b) = f(g(Y), W)$.

Proseguendo con la seconda equazione dell'insieme originario, opportunamente istanziata dalle sostituzioni sin qui calcolate, dall'operazione (i) otteniamo

$$E_4 = \{X = g(Y), W = b, g(Y) = Z, Y = b\};$$

quindi da (vi) otteniamo

$$E_5 = \{X = g(Y), W = b, Z = g(Y), Y = b\}.$$

Infine da (iv), applicata all'ultima equazione, abbiamo

$$E_4 = \{X = g(b), W = b, Z = g(b), Y = b\},$$

che costituisce la forma risolta dell'insieme E e quindi fornisce anche l'm.g.u. dell'insieme iniziale, nella forma della sostituzione

$$\vartheta = \{X/g(b), W/b, Z/g(b), Y/b\}.$$

Ci sono due osservazioni, importanti, da fare. Innanzitutto si noti come il valore di alcune variabili possa essere prima specificato parzialmente, per poi essere ulteriormente definito in un tempo successivo. Ad esempio, ϑ_1 (m.g.u. della prima equazione di E) ci dice che X ha come valore il termine $g(Y)$ e solo risolvendo anche la seconda equazione vediamo che Y ha come valore b , e quindi che X ha valore $g(b)$ (come in effetti risulta nell'm.g.u. finale).

Inoltre, osserviamo che se consideriamo $\{h(g(X), Y) = h(Z, W)\}\vartheta_1$ (la seconda equazione di E istanziata dall'm.g.u. della prima), otteniamo l'equazione

$$h(g(Y), Y) = h(Z, b)$$

per la quale la sostituzione

$$\vartheta_2 = \{Z = g(b), Y = b\}$$

è un m.g.u. Usando la definizione di composizione di sostituzione è facile verificare che $\vartheta = \vartheta_1\vartheta_2$: l'm.g.u. dell'insieme E si può dunque ottenere componendo l'm.g.u. della prima equazione con quello della seconda, alla quale sia applicato il primo m.g.u. Questo, come già dicevamo, è quello che in effetti avviene normalmente nelle implementazioni dei linguaggi logici, dove invece che accumulare tutte le equazioni per poi risolverle alla fine, ad ogni passo di computazione viene calcolato un m.g.u. e viene composto con quelli precedentemente ottenuti.

14.4 Il modello computazionale

Il paradigma logico, realizzando l'idea di "computazione come deduzione", ha un modello computazionale sostanzialmente diverso da tutti quelli che abbiamo sin qui visto. Volendo sintetizzare, possiamo individuare le seguenti differenze principali rispetto agli altri paradigmi.

1. Gli unici valori possibili, almeno nel modello puro, sono i termini su una data segnatura.
2. I programmi possono avere una lettura dichiarativa, interamente logica, o una lettura procedurale di tipo più operazionale.
3. La computazione avviene istanziando le variabili che appaiono nei termini (e quindi nei goal) con altri termini, usando il meccanismo dell'unificazione.
4. Il controllo, interamente gestito dalla macchina astratta (salvo alcune annotazioni possibili in PROLOG) è basato sul meccanismo del backtracking automatico.

Nel seguito cercheremo dunque di illustrare il modello computazionale del paradigma logico analizzando questi quattro punti. Discuteremo esplicitamente le differenze fra programmazione logica e PROLOG.

14.4.1 L'universo di Herbrand

Nell'ambito della programmazione logica i termini costituiscono un elemento fondamentale. L'insieme di tutti i possibili termini su una data segnatura è detto universo di Herbrand e costituisce il dominio sul quale viene effettuata la computazione da un programma logico. Vi sono alcune particolarità da notare al riguardo.

- Relativamente ai simboli non logici l'alfabeto sul quale sono definiti i programmi non è prefissato ma può variare. Spesso è determinato dai simboli che compaiono nel particolare programma in questione, ma altre volte contiene anche altri simboli.
- Come (parziale) conseguenza del precedente punto, a differenza di quanto avviene nei linguaggi imperativi, nessun significato predefinito è associato ai simboli (non logici) dell'alfabeto. Ad esempio, un programma può usare il simbolo $+$ per denotare la somma aritmetica mentre un altro con tale simbolo può indicare la concatenazione di stringhe. Fanno eccezione il predicato (binario) d'uguaglianza, alcuni altri predicati detti "built-in" presenti in PROLOG¹⁷.
- Come ulteriore conseguenza, nessun sistema di tipi è presente nei linguaggi logici (almeno nel formalismo classico): l'unico tipo presente è quello dei termini con i quali possiamo rappresentare espressioni aritmetiche, liste ecc.

Dal punto di vista teorico, il fatto che non vi siano tipi e che la computazione avvenga sull'universo di Herbrand non è limitativo, anzi permette di esprimere in modo elegante e, tutto sommato, semplice, la semantica formale dei programmi logici. Ad esempio, già con due soli simboli di funzione, 0 (costante zero) e s (successore, di arietà 1) riusciamo ad esprimere i numeri naturali con i termini $0, s(0), s(s(0)), s(s(s(0)))$ ecc. Con qualche attenzione possiamo esprimere le normali operazioni aritmetiche in termini di questa rappresentazione a due simboli.

Dal punto di vista pratico, invece, la mancanza di tipi è un problema serio e difatti, sia in PROLOG che in alcuni linguaggi logici più recenti sono stati introdotti alcuni tipi elementari (ad esempio, gli interi con le relative operazioni aritmetiche). I linguaggi di questo paradigma risultano comunque sempre carenti dal punto di vista dei tipi.

14.4.2 Interpretazione dichiarativa e procedurale

Come abbiamo già accennato, una clausola, e quindi un programma logico, può avere due interpretazioni diverse: una *dichiarativa* ed una *procedurale*.

Dal punto di vista *dichiarativo*, una clausola $H : -A_1, \dots, A_n$. è una formula che, sostanzialmente, esprime che se A_1 e $A_2 \dots$ e A_n sono veri allora è vero anche H . Una query (o goal) è anch'essa una formula per la quale vogliamo dimostrare che, se opportunamente istanziata, è una conseguenza logica del programma, ossia vale in tutte le interpretazioni nelle quali vale il programma¹⁸. Questa interpretazione può essere sviluppata, usando gli strumenti della logica (in particolare alcune nozioni elementari di teoria dei modelli) in modo tale da dare un significato ad un programma in termini puramente dichiarativi, senza alcun riferimento ad un processo di calcolo. Per questa interpretazione, per altro interessante, rimandiamo alla letteratura specializzata citata alla fine del capitolo.

¹⁷ Ed anche i predicati predefiniti nei linguaggi con vincoli, che però noi qui non consideriamo.

¹⁸ Ci accontentiamo qui di un'idea intuitiva di questi concetti. Il lettore interessato può vedere una qualsiasi testo di logica per maggiori dettagli.

L'interpretazione *procedurale*, invece, permette di leggere una clausola

$$H : -A_1, \dots, A_n.$$

come segue: per dimostrare H devi prima dimostrare A_1, \dots, A_n , o anche, per calcolare H devi prima calcolare A_1, \dots, A_n . In questo senso possiamo vedere un predicato come un nome di procedura: le clausole che lo definiscono costituiscono il suo corpo. In questa interpretazione possiamo leggere un atomo nel corpo di una clausola, o in un goal, come una chiamata di procedura. Un programma logico è dunque un insieme di dichiarazioni e un goal non è altro che l'equivalente del "main" di un programma imperativo, dato che contiene tutte le chiamate di procedura che si vogliono valutare. La virgola nel corpo delle clausole e nei goal, in PROLOG (ma non nella programmazione logica pura), può essere vista come l'analogo del ";" dei linguaggi imperativi.

Precisi teoremi di corrispondenza permettono di riconciliare la visione dichiarativa e quella procedurale, dimostrando che i due approcci sono equivalenti.

Da un punto di vista formale, l'interpretazione procedurale è supportata dalla cosiddetta risoluzione SLD, una regola di derivazione logica che discuteremo nell'Approfondimento 14.1. Ma è anche possibile descrivere l'interpretazione procedurale in modo più informale, usando solo l'analogia che abbiamo appena espresso con le chiamate di procedura ed il passaggio dei parametri. È questo l'approccio che useremo nel seguito.

14.4.3 La chiamata di procedura

Consideriamo per il momento una definizione semplificata di clausola, dove assumiamo che nella testa tutti gli argomenti del predicato siano variabili distinte. Una generica clausola di questo tipo ha dunque la forma

$$p(X_1, \dots, X_n) : -A_1, \dots, A_m.$$

e, come abbiamo anticipato, può essere vista come la dichiarazione della procedura p con n parametri formali X_1, \dots, X_n . Un atomo $q(t_1, \dots, t_n)$ può essere visto come la chiamata della procedura q con gli n parametri attuali t_1, \dots, t_n . Nella definizione di p , dunque, il corpo è costituito dall'invocazione delle m procedure che costituiscono gli atomi A_1, \dots, A_m .

In accordo a questa visione e analogamente a quanto avviene nei linguaggi imperativi, la valutazione della chiamata $p(t_1, \dots, t_n)$ causa la valutazione del corpo della procedura, dopo aver effettuato il passaggio dei parametri. Questo avviene con una modalità simile al passaggio per nome, rimpiazzando nel corpo della procedura il parametro formale X_i con il corrispondente attuale t_i . Inoltre, dato che le variabili che compaiono nel corpo della procedura sono da considerarsi locali, esse devono essere considerate distinte da tutte le altre variabili. Nei linguaggi a blocchi questo avviene implicitamente, dato che il corpo della procedura è considerato come un blocco con un suo ambiente locale. Qui invece, non essendo presente la nozione di blocco, per evitare possibili conflitti tra i nomi di variabile, supponiamo che, prima di usare una clausola, tutte le variabili

che occorrono in essa siano state ridenominate opportunamente in modo da essere diverse da tutte le altre variabili presenti.

Riassumendo in termini più precisi quanto detto sopra, possiamo dire che la valutazione della chiamata $p(t_1, \dots, t_n)$, con la definizione di p vista poc' anzi, causa la valutazione delle m chiamate di procedura

$$(A_1, \dots, A_m)\vartheta$$

presenti nel corpo di p , opportunamente istanziate dalla sostituzione

$$\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$$

che effettua il passaggio dei parametri. Nel caso in cui il corpo della clausola sia vuoto (ossia $m = 0$), la chiamata di procedura termina immediatamente. Altrimenti la computazione prosegue con la valutazione delle nuove chiamate. Usando la terminologia della programmazione logica, questo si esprime dicendo che la valutazione del goal $p(t_1, \dots, t_n)$ produce il nuovo goal

$$(A_1, \dots, A_m)\{X_1/t_1, \dots, X_n/t_n\}$$

che, a sua volta, dovrà essere valutato. Quando tutte le chiamate generate con questo processo sono state valutate (senza che vi sia stato alcun fallimento) la computazione termina con successo e il risultato finale è costituito dalla sostituzione che associa alle variabili presenti nella chiamata iniziale (X_1, \dots, X_n nel nostro caso) i valori calcolati nel corso della computazione. Questa sostituzione è detta *risposta calcolata* per il goal iniziale nel programma dato. Vedremo una definizione più precisa di questa nozione più avanti, per ora vediamo un semplice esempio. Consideriamo la procedura `lista_di_2` definita qui sotto ed identica alla procedura `lista_di_27` del Paragrafo 14.1.1, salvo il minor numero di variabili anonime.

```
lista_di_2(Ls) :- Ls = [_,_].
```

La valutazione della chiamata `lista_di_2(Lxs)` causa la valutazione del corpo della clausola, istanziato dalla sostituzione $\{Ls/Lxs\}$, e cioè

```
Lxs = [_,_]
```

Questa è una chiamata particolare, perché $=$ è un predicato predefinito che, come abbiamo visto nel paragrafo precedente, è interpretato come uguaglianza sintattica sull'universo di Herbrand e operazionalmente corrisponde al processo di unificazione. La chiamata precedente dunque si riduce al tentativo (fatto dall'interprete del linguaggio) di risolvere l'equazione usando il meccanismo dell'unificazione. Nel nostro caso, ovviamente, tale tentativo ha successo e produce l'm.g.u. $\{Lxs/[_,_]\}$ che è il risultato della computazione: la precedente sostituzione è dunque risposta calcolata per il goal `lista_di_2` nel programma logico costituito dall'unica linea (1) sopra.

Valutazione di un goal non atomico Quanto visto nel paragrafo precedente deve essere generalizzato e precisato meglio perché sia chiaro il modello computazionale del paradigma logico. Nel seguito, per conformarci alla terminologia corrente, parleremo di goal atomico e di goal, invece che di chiamata di procedura e di sequenza di chiamate. Rimane valida tuttavia l'analogia con le procedure dei paradigmi convenzionali.

Nel caso in cui si debba valutare un goal non atomico il meccanismo computazionale è analogo a quello visto sopra, salvo che adesso dobbiamo selezionare una delle chiamate possibili usando un'opportuna *regola di selezione*. Mentre nel caso della programmazione logica pura non è specificata alcuna regola, PROLOG adotta la regola che seleziona sempre l'atomo più a sinistra. È comunque possibile dimostrare che le risposte calcolate sono sempre le stesse, indipendentemente da quale sia la regola adottata (si vedano comunque anche gli Esercizi 13 e 14 alla fine del capitolo).

Supponendo, per semplicità, di adottare la regola di PROLOG, possiamo descrivere come segue il processo di valutazione. Sia

$$B_1, \dots, B_k$$

con $k \geq 1$ il goal da valutare. Distinguiamo i seguenti casi a seconda della forma dell'atomo selezionato B_1 .

1. se B_1 è un'equazione della forma $s = t$ allora si cerca di calcolarne un m.g.u. (usando l'algoritmo di unificazione). Abbiamo due possibilità:

(a) se l'm.g.u. esiste ed è la sostituzione σ , allora il risultato della valutazione è il goal

$$(B_2, \dots, B_k)\sigma$$

ottenuto dal precedente eliminando l'atomo selezionato e applicando l'm.g.u. calcolato. Se $k = 1$ (e dunque $(B_2, \dots, B_k)\sigma$ è vuoto), la computazione termina con successo.

(b) se l'm.g.u. non esiste (ossia l'equazione non ha soluzioni) allora si ha un fallimento;

2. se invece B_1 ha la forma $p(t_1, \dots, t_n)$ abbiamo i seguenti due casi:

(a) se esiste nel programma una clausola della forma

$$p(X_1, \dots, X_n) : -A_1, \dots, A_m.$$

(che consideriamo ridenominata per evitare cattura di variabili), allora il risultato della valutazione è un nuovo goal

$$(A_1, \dots, A_m)\vartheta, B_2, \dots, B_k$$

dove $\vartheta = \{X_1/t_1, \dots, X_n/t_n\}$. Se $k = 1$ (ossia avevamo un goal atomico) e $m = 0$ (il corpo della clausola è vuoto) allora la computazione termina con successo;

(b) se nel programma non esiste alcuna clausola che definisce il predicato p , allora si ha un fallimento.

Per poter definire con esattezza i risultati delle computazioni (le risposte calcolate) abbiamo bisogno di chiarire alcuni aspetti relativi al controllo, cosa che facciamo nel Paragrafo 14.4.4.

Teste con termini generici Sin qui abbiamo supposto che le teste delle clausole contenessero solo variabili distinte. Abbiamo fatto questa scelta per conservare la similarità con le procedure dei linguaggi tradizionali, tuttavia i programmi logici reali usano anche termini generici come argomenti dei predicati nelle teste, come abbiamo visto nell'esempio del Paragrafo 14.1.1. L'Approfondimento 14.1 fornisce la regola di valutazione per tale caso più generale. Si noti, comunque, che la nostra assunzione non è in alcun modo limitativa (a parte, forse, la convenienza grafica): infatti, come appare evidente da quanto detto nel riquadro e dalla precedente trattazione, una clausola della forma

$$p(t_1, \dots, t_n) : -A_1, \dots, A_m.$$

può essere vista come un'abbreviazione per la clausola

$$p(X_1, \dots, X_n) : -X_1 = t_1, \dots, X_n = t_n, A_1, \dots, A_m.$$

Negli esempi seguenti useremo spesso la notazione con generici termini nelle teste.

Approfondimento 14.1

14.4.4 Controllo: il non determinismo

Nella valutazione di un goal abbiamo due gradi di libertà: la selezione dell'atomo da valutare e la scelta della clausola da applicare.

Per il primo abbiamo detto che possiamo fissare una regola di selezione, senza che questo influenzi i risultati finali delle computazioni che terminano con successo.

Per la scelta della clausola invece la cosa è più delicata. Dato che un predicato può essere definito da più clausole e dobbiamo usarne una sola alla volta, potremmo pensare di fissare una qualche regola di scelta delle clausole, analogamente a quanto fatto con la selezione degli atomi. Il seguente esempio mostra però un problema. Consideriamo il seguente programma, che chiamiamo P_a :

```
p(X) :- p(X).
p(X) :- X=a.
```

e supponiamo di scegliere le clausole dall'alto in basso, seguendo l'ordine testuale del programma. È facile vedere che adottando tale regola, la valutazione del goal $p(Y)$ non termina mai e quindi non abbiamo alcuna risposta calcolata dal programma. Infatti, usando la clausola (1) e la sostituzione $\{X/Y\}$, il goal iniziale, dopo un passo di computazione, diviene il goal $p(X)\{X/Y\}$ che è di nuovo il goal di partenza. A questo goal, secondo la regola di scelta della clausola, dobbiamo applicare di nuovo la clausola (1) e così via. È però evidente che usando la clausola (2) otterremmo immediatamente una computazione che termina, producendo come risposta la sostituzione $\{Y/a\}$. Si noti che fissare un altro ordine, ad esempio dal basso verso l'alto, non risolve in generale il problema. Alla luce di questo esempio riconsideriamo con attenzione la regola di valutazione di un goal vista nel precedente paragrafo e, in particolare, soffermiamoci sul punto 2(a), dove sta scritto "se esiste una clausola" senza specificare come questa debba essere scelta. Così facendo abbiamo dunque introdotto nel modello di computazione una forma di non determinismo: nel caso in cui vi siano più clausole per lo stesso predicato dobbiamo sceglierne una in modo non deterministico, senza fissare alcuna regola. Questa forma di non-determinismo è detta "*don't know*", in quanto non sappiamo qual è la clausola "giusta" che ci permette di terminare con successo la computazione. Il modello teorico della programmazione logica mantiene questo non-determinismo, in quanto vengono considerate tutte le possibili scelte della clausole e quindi tutti i possibili risultati delle varie computazioni prodotte in conseguenza di tali scelte. Il risultato della valutazione di un goal G nel programma P è dunque un insieme di risposte calcolate, dove ognuna di tali risposte è la sostituzione ottenuta dalla composizione di tutti gli m.g.u. che si incontrano in una specifica computazione (con specifiche scelte di clausole), ristretta alle variabili presenti in G . Per una definizione più precisa di questa nozione, così come dell'intero processo di valutazione dei goal, si veda l'Approfondimento 14.1.

Tornando al nostro esempio precedente, l'unica risposta calcolata per il goal $p(Y)$ nel programma P_a è la sostituzione $\{Y/a\}$ mentre, sempre per lo stesso programma, il goal $p(b)$ non ha alcuna risposta calcolata dato che tutte le sue computazioni o terminano con fallimento (quando si usa seconda clausola) oppure non terminano (quando si usa solo la prima clausola).

Il backtracking in PROLOG Quando dal modello teorico si passa ad un linguaggio implementato, quale PROLOG, il non-determinismo ad un qualche livello deve essere trasformato in determinismo, dato che le macchine fisiche che usiamo come calcolatori sono deterministiche. Questo può ovviamente essere fatto in vari modi e in linea di principio non causa la perdita di soluzioni: ad esempio, potremmo pensare di far partire k computazioni parallele quando si abbiano k possibili clausole per un predicato¹⁹ e quindi considerare i risultati di tutte le possibili computazioni.

¹⁹Ovviamente, su una macchina con un solo processore, le k computazioni parallele devono essere opportunamente "scheduled" per poter essere eseguite in modo sequenziale, analogamente a quanto avviene con i processi in un sistema operativo multitasking.

In PROLOG tuttavia, per motivi di semplicità e di efficienza dell'implementazione, viene usata la strategia vista prima: le clausole vengono usate secondo l'ordine testuale con cui appaiono nel programma (dall'alto in basso). Abbiamo visto con il precedente esempio che questa strategia è *incompleta*, dato che non permette di trovare tutte le possibili risposte calcolate. Questo limite tuttavia è aggirabile dal programmatore che, conoscendo questa caratteristica di PROLOG, può ordinare le clausole del programma nel modo più conveniente (tipicamente mettendo prima quelle relative ai casi terminali e poi quelle induttive). Si noti però che questo, almeno in linea di principio, elimina parte della dichiaratività del linguaggio, in quanto al programmatore è richiesta la specifica di un aspetto di controllo.

Oltre alle computazioni infinite, vi è un secondo aspetto, più importante, da considerare adottando il modello deterministico di PROLOG e riguarda la gestione dei fallimenti. Vediamo prima un esempio. Consideriamo il programma P_b

```
p(X) :- X=f(a).
p(X) :- X=g(a).
```

e consideriamo la valutazione del goal $p(g(Y))$. Secondo la strategia di PROLOG viene scelta la clausola (1), che (usando la sostituzione $\{X/g(Y)\}$) da luogo al nuovo goal $g(Y) = f(a)$. Questo fallisce, dato che i due termini dell'equazione non sono unificabili. Tuttavia, dato che vi è ancora una clausola da usare, non sarebbe accettabile terminare la computazione riportando un fallimento. Viene dunque fatto un "backtracking" tornando alla scelta della clausola per $p(g(Y))$ e quindi si prosegue la computazione provando la clausola (2). In questo modo si arriva al successo con risposta calcolata $\{Y/a\}$.

In generale dunque, quando si arriva ad un fallimento, la macchina astratta PROLOG fa "backtracking" fino al precedente punto di scelta nel quale vi siano altre possibilità, ossia altre clausole da provare. In questo processo di backtracking gli eventuali legami che erano stati calcolati nella computazione precedente devono essere disfatti. Giunti al punto di scelta, viene provata una nuova clausola e la computazione prosegue nel modo visto. Se al precedente punto di scelta non vi sono altre possibilità, si risale al punto di scelta ancora precedente e se non ve ne sono la computazione termina con un fallimento. Si noti che tutto questo è gestito direttamente dalla macchina astratta PROLOG ed è completamente invisibile al programmatore (salvo l'uso di costrutti particolari, quali il cut che introdurremo nel Paragrafo 14.5.1).

È anche facile rendersi conto come questo procedere per tentativi, che sostanzialmente corrisponde ad una ricerca in profondità in un albero che rappresenta tutte le possibili computazioni, può essere molto pesante dal punto di vista computazionale. La soluzione del problema visto nel Paragrafo 14.1.1, ad esempio, richiede un uso esteso del backtracking ed è abbastanza dispendiosa dal punto di vista del tempo di calcolo.

Vediamo un altro esempio. Consideriamo il programma P_c

```
p(X) :- X=f(Y), q(X).
p(X) :- X=g(Y), q(X).
q(X) :- X=g(a).
```

$q(X) :- x = g(b).$

ed analizziamo la valutazione del goal $p(Z)$. Usando la clausola (1) otteniamo il goal $z = f(Y), q(Z)$; la valutazione dell'equazione produce l'm.g.u. $\{Z/f(Y)\}$ e rimane il goal $q(f(Y))$. Usando la clausola (3) otteniamo il goal $f(Y) = g(a)$, che fallisce. A questo punto dobbiamo tornare²⁰ al precedente punto di scelta, ossia alla scelta della clausola per il predicato q . In questo caso non vi sono legami da "disfare" e dunque proviamo la clausola (4), ottenendo così il goal $f(Y) = g(b)$ che fallisce anch'esso. Torniamo di nuovo al punto di scelta per q , vediamo che non vi sono altre possibili clausole e dunque torniamo al precedente punto di scelta che è quello per il predicato p . Facendo questo dobbiamo disfare il legame $\{Z/f(Y)\}$ che era stato calcolato dalla clausola (1) e dunque ritorniamo alla situazione iniziale, dove la variabile z non è istanziata. Usando la clausola (2) otteniamo il goal $z = g(Y), q(Z)$ e quindi, dalla valutazione dell'equazione, otteniamo l'm.g.u. $\{Z/g(Y)\}$ ed il nuovo goal $q(g(Y))$. A questo punto, usando la clausola (3) otteniamo il goal $g(Y) = g(a)$ che ha successo e produce l'm.g.u. $\{Y/a\}$. Dato che non rimangono altri goal da valutare, la computazione termina con successo. Il risultato della computazione è prodotto componendo gli m.g.u. calcolati $\{Z/g(Y)\}\{Y/a\}$ e restringendo la sostituzione $\{Z/g(a), Y/a\}$ ottenuta da tale composizione all'unica variabile presente nel goal iniziale: otteniamo così la risposta calcolata $\{Z/g(a)\}$. Si noti che vi è un'altra risposta calcolata $\{Z/g(b)\}$, che possiamo ottenere usando la clausola (4) invece che la (3). Nelle implementazioni PROLOG si possono ottenere le risposte successive alla prima usando il comando ";". Infine, il lettore può facilmente verificare che il goal $p(g(c))$ nel programma PC termina con un fallimento, ottenuto dopo aver provato tutte e quattro le combinazioni di clausole possibili.

14.4.5 Alcuni esempi

In questo paragrafo faremo riferimento al linguaggio PROLOG, del quale seguiremo anche la sintassi. La notazione $[h \mid t]$ è usata per indicare la lista che ha come testa h e come coda t . Ricordiamo che la testa è il primo elemento della lista, mentre la coda è la lista costituita dai rimanenti elementi, una volta tolto il primo. La lista vuota è denotata da $[]$ mentre la scrittura $[a, b, c]$ è un'abbreviazione per $[a \mid [b \mid [c \mid []]]]$ (la lista costituita dai tre elementi a , b e c). Si noti che in PROLOG, così come nella programmazione logica pura, non esiste il tipo lista, per cui il simbolo di funzione binaria $[\mid]$ può essere usato anche per termini che non sono liste: ad esempio, possiamo anche scrivere $[a \mid f(a)]$ che non è una lista (perché $f(a)$ non è una lista).

Come primo esempio consideriamo il seguente programma MEMBER che verifica se un elemento appartiene ad una data lista:

`member(X, [X | Xs]).`

²⁰Usiamo il plurale per convenzione antropomorfa: tutto questo è fatto, ovviamente, dalla macchina astratta PROLOG.

```
member(X, [Y | Xs]) :- member(X, Xs).
```

La lettura dichiarativa del precedente programma è immediata: la clausola (1) costituisce il caso terminale, in cui l'elemento che stiamo cercando (il primo argomento del predicato `member`) è la testa della lista che abbiamo (il secondo argomento di `member`). La clausola (2) fornisce invece il caso induttivo e ci dice che `X` è un elemento della lista `[Y | Xs]` se è un elemento della lista `Xs`.²

Così formulato, il programma MEMBER è simile a quello che possiamo scrivere in un qualsiasi linguaggio che supporti la ricorsione. Tuttavia notiamo che, a differenza di quanto avviene nei paradigmi funzionale e imperativo, il precedente programma può essere usato in vari modi diversi.

Il modo più convenzionale è quello di usarlo come test: in un sistema PROLOG, una volta immesso il precedente programma, abbiamo

```
?- member(pippo, [pluto, pippo, ciccio]).
```

Yes

dove `?-` è il prompt della macchina astratta, al quale abbiamo fatto seguire il goal di cui si chiede la valutazione. La linea successiva riporta la risposta dell'interprete. In questo caso abbiamo una semplice risposta "booleana", che esprime l'esistenza di una computazione di successo per il nostro goal. Tuttavia, come sappiamo, possiamo anche usare il programma per calcolare. Ad esempio possiamo chiedere la valutazione di

```
?- member(X, [pluto, pippo, ciccio]).
```

`X = pluto`

La macchina astratta restituisce `{X/pluto}` come una risposta calcolata. Possiamo anche ottenere le riposte successive usando il comando `";"`. Quando non vi sono più risposte il sistema risponde "no".

Infine, anche se si tratta di un uso poco naturale, possiamo usare il primo argomento per istanziare la lista nel secondo argomento. Ad esempio abbiamo

```
?- member(pluto, [X, pippo, ciccio]).
```

`X = pluto`

Questa possibilità di usare gli stessi argomenti come input o output, a seconda di come sono istanziati, è unica del paradigma logico ed è dovuta alla presenza del meccanismo di unificazione nel modello di calcolo.

Possiamo chiarire ulteriormente questo punto considerando il seguente programma APPEND che permette di concatenare due liste.¹

```
append([], Ys, Ys).
```

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Anche in questo caso la lettura dichiarativa è immediata: se la prima lista è vuota, allora il risultato è costituito dalla seconda lista (clausola (1)). Altrimenti (induttivamente) se `Zs` è il risultato della concatenazione di `Xs` e `Ys`, il risultato della concatenazione di `[X|Xs]` e `Ys` è ottenuto aggiungendo `X` in testa alla lista `Zs`, come indicato in `[X|Zs]`.

L'uso normale del precedente programma è quello illustrato dal seguente goal

```
?- append([pluto, pippo], [ciccio, qui], Zs).
Zs = [pluto, pippo, ciccio, qui]
```

Tuttavia, possiamo usare APPEND anche per sapere come possiamo suddividere una lista in due sottoliste, cosa che non è possibile con una versione funzionale o imperativa del programma:

```
?- append(Xs, Ys, [pluto, pippo]).
```

```
Xs = []
Ys = [pluto, pippo];
Xs = [pluto]
Ys = [pippo];
Xs = [pluto, pippo]
Ys = [];
no
```

dove, come ricordato poc' anzi, il comando ";" permette di ottenere nuove soluzioni.

Come terzo esempio diamo la definizione del predicato sublist, che abbiamo usato nel Paragrafo 14.1.1. Se Xs è una sottolista di Ys allora esistono altre due liste (possibilmente vuote) As e Bs tali che Ys è la concatenazione di As, Xs e Bs. Osserviamo che questo significa che Xs è il suffisso di un prefisso di Ys per cui, usando APPEND, possiamo definire sublist come segue:

```
sublist(Xs, Ys) :- append(As, XsBs, Ys), append(Xs, Bs, XsBs).
```

Il precedente programma, unitamente al programma APPEND e al programma SEQUENZA del Paragrafo 14.1.1, permette di risolvere il problema enunciato nel Paragrafo 14.1.1 stesso. Si noti la concisione e semplicità del programma risultante. Ovviamente altre definizioni di sublist sono possibili (si veda l'esercizio 8).

Come ultimo esempio consideriamo un programma che risolve il classico problema delle torri di Hanoi. Si ha una torre (in termini informatici diremmo una pila) costituita da n dischi forati (di diametri diversi), disposti su di un'asta in ordine di diametro decrescente e si hanno altre due aste vuote. Il problema consiste nello spostare la torre ad un'altra asta, ricreando l'ordinamento iniziale dei dischi e rispettando le seguenti regole: i dischi possono essere spostati solo da un'asta ad un'altra asta; non si può spostare più di un disco alla volta da un'asta all'altra e si deve prelevare sempre il disco in cima ad un'asta; un disco non può mai essere posto sopra un altro disco di diametro più piccolo.

Secondo la leggenda, questo problema fu assegnato dalla Divinità ai monaci di un monastero nei pressi di Hanoi, con tre aste e 64 dischi d'oro. La soluzione del problema avrebbe segnato la fine del mondo. Dato che la soluzione ottima richiede un tempo esponenziale nel numero dei dischi, anche se la leggenda si dovesse avverare possiamo stare tranquilli ancora per un bel pezzo: 2^{64} è un numero sufficientemente grande.

Il seguente programma risolve il problema per un generico numero di dischi N e tre aste che chiamiamo A, B e C. Usiamo la codifica dei numeri naturali in termini di 0 e di suoi successori, già vista in precedenza.

```
hanoi(s(0), A, B, C [sposta(A, B)]).
```

14.5.3 Programmazione logica con vincoli

Per concludere accenniamo brevemente alla programmazione logica con vincoli, o CLP (*Constraint Logic Programming*), un'estensione della programmazione logica che unisce al modello computazionale già visto meccanismi di soluzione di vincoli, anche molto sofisticati. Il paradigma risultante è di sicuro interesse per le applicazioni pratiche ed è oggi usato in vari campi applicativi.

Un vincolo (*constraint* in inglese) non è altro che una particolare formula della logica del prim'ordine (normalmente si tratta di una congiunzione di formule atomiche) che usa solo predicati di significato predefinito. Un esempio di vincolo lo abbiamo già incontrato: se scriviamo in un programma logico $x=t$ o anche $x=t, Y = f(a)$ abbiamo usato dei vincoli, dove il simbolo di predicato $=$, come sappiamo, è interpretato come uguaglianza sintattica sull'universo di Herbrand²⁷ mentre la virgola indica la congiunzione logica.

L'idea della programmazione logica con vincoli è quella di rimpiazzare l'universo di Herbrand con un diverso dominio di computazione, simbolico ma anche numerico in alcuni casi, adatto alla specifica classe di applicazioni che interessa. I vincoli, invece che relazioni fra termini ground, definiscono relazioni sui valori del nuovo dominio considerato. Corrispondentemente, il meccanismo di calcolo di base non sarà più la soluzione di equazioni fra termini (mediante unificazione), ma sarà costituito da uno specifico risolutore di vincoli, ossia da un opportuno algoritmo per determinare la soluzione dei vincoli considerati. Ad esempio, potremmo considerare come dominio della computazione i numeri reali, come vincoli congiunzioni di equazioni lineari, e come risolutore potremmo usare il metodo di Gauss-Jordan.

Il vantaggio principale di questo approccio è evidente: possiamo integrare nel paradigma logico (in modo, fra l'altro, semanticamente molto pulito) meccanismi di calcolo molto potenti sviluppati anche in altri contesti (quali la programmazione lineare, la ricerca operativa ecc.). Domini particolarmente interessanti per

²⁷Ricordiamo che questo è l'insieme dei termini ground.

le applicazioni pratiche sono i già menzionati numeri reali²⁸ e, soprattutto, i domini finiti, nei quali le variabili possono assumere un numero finito di valori. In quest'ultimo caso si parla anche di problemi soddisfacibilità di vincoli o CSP (*Constraint Satisfaction Problem*) ed esistono numerosi algoritmi specifici per la soluzione di questo tipo di problemi.

Per avere un'idea delle possibilità della programmazione logica con vincoli si consideri il problema di definire l'ammontare delle rate di un mutuo. Le quantità coinvolte in questo problema sono le seguenti: Fin è il finanziamento richiesto, ossia la somma iniziale presa in prestito; NumR è il numero delle rate; Int è il tasso d'interesse; Rata è l'importo della singola rata; infine Deb è il debito rimanente. La relazione fra queste variabili è intuitivamente la seguente. Nel caso in cui non vi sia alcuna rata rimborsata (ossia NumR = 0) evidentemente abbiamo che il debito è eguale al finanziamento iniziale:

$$\text{Deb} = \text{Fin}.$$

Nel caso del pagamento di una singola rata invece abbiamo la seguente relazione:

$$\text{Deb} = \text{Fin} + \text{Fin} * \text{Int} - \text{Rata}$$

ossia il debito rimanente è l'importo iniziale al quale abbiamo aggiunto gli interessi passivi maturati e sottratto quanto rimborsato con una rata. Nel caso di due rate la cosa è più complicata perché gli interessi vanno calcolati sul debito rimanente e non sull'importo iniziale. Il debito rimanente, cioè, diventa il nuovo valore del finanziamento. Usando le variabili NuFin1 e NuFin2 per indicare tale nuovo valore del finanziamento, in questo caso abbiamo dunque le relazioni

$$\begin{aligned}\text{NuFin1} &= \text{Fin} + \text{Fin} * \text{Int} - \text{Rata} \\ \text{NuFin2} &= \text{NuFin1} + \text{Fin} * \text{Int} - \text{Rata} \\ \text{Deb} &= \text{NuFin2}\end{aligned}$$

Nel caso generale, al solito, possiamo ragionare ricorsivamente. Abbiamo dunque il seguente programma con vincoli che chiamiamo MUTUO

```
mutuo(Fin, NumR, Int, Rata, Deb) :-  
    NumR = 0,  
    Deb = Fin.
```

```
mutuo(Fin, NumR, Int, Rata, Deb) :-  
    NumR >= 1,  
    NuFin = Fin+Fin*Int-Rata,  
    NuNumR = NumR-1,  
    mutuo(NuFin, NuNumR, Int, Rata, Deb).
```

la cui comprensione, alla luce di quanto detto sopra, non dovrebbe porre difficoltà.

Un tale programma può essere usato in molti modi, analogamente a quanto avviene con i programmi logici. Ad esempio, possiamo usarlo per sapere qual è il debito rimanente, avendo preso un finanziamento di 1000 euro e avendo pagato

²⁸Ovviamente, come sempre quando si tratta di calcolatori, quello che viene effettivamente rappresentato è un sottoinsieme dei reali.

10 rate di 150 euro l'una, con un tasso d'interesse del 10%. La query per risolvere questo problema è

```
mutuo(1000, 10, 10/100, 150, Deb)
```

che se valutata produce il risultato $Deb = 203.13$.

Possiamo anche usare lo stesso programma in "senso inverso", per così dire, ossia per sapere qual è il finanziamento che avevamo chiesto, sapendo che abbiamo pagato le stesse 10 rate di 150 euro l'una di prima, sempre con un tasso d'interesse del 10%, ma con debito residuo 0. Il goal in questo caso è

```
mutuo(Fin, 10, 10/100, 150, 0)
```

che dà il risultato $Fin = 921.685$.

Infine potremmo anche lasciare più di una variabile nel goal, ad esempio per sapere la relazione che esiste fra il finanziamento, la rata e il debito, sapendo che paghiamo 10 rate con tasso d'interesse del 10%. Possiamo quindi formulare la query

```
mutuo(Fin, 10, 10/100, Rata, Deb)
```

che (con opportune ipotesi sul risolutore²⁹) produce il risultato

$$Fin = 0.3855 * Deb + 6.1446 * Rata.$$

ossia, cosa unica nei vari paradigmi sin qui visti, permette di ottenere, come risultato, una relazione su domini numerici.