5

PROGRAMMAZIONE LOGICA: INTRODUZIONE ED ESEMPI

- la programmazione logica si basa su particolari teorie del primo ordine, i cui assiomi sono

  clausole Horn definite

- tali teorie hanno delle proprietà particolari

  - dal punto di vista proof-theoretic

    "la input resolution è completa, --, la SLD-resolution è completa"

  - dal punto di vista model-theoretic

    "esiste un modello di Herbrand canonico"

- tali proprietà mostrano che le clausole Horn definite sono un vero e proprio linguaggio di programmazione

LE CLAUSOLE HORN

- una generica clausola

$$A_1 \vee \dots \vee A_K \vee \neg B_1 \vee \dots \vee \neg B_n$$

può essere equivalentemente rappresentata nella forma

conclusioni — premesse

$$A_1, \dots, A_K \leftarrow B_1, \dots, B_n$$

da leggere

$$(B_1 \wedge \dots \wedge B_n) \supset (A_1 \vee \dots \vee A_K)$$

(in ambedue le forme, è implicita l'esistenza di un prefisso in cui tutte le variabili sono quantificate universalmente).

- clausole Horn: $K \leq 1$ (al max una conclusione)

  - clausole definite (o di programma), se $K = 1$.

$$A_1 \leftarrow B_1, \dots, B_n$$

    - (un caso particolare) clausole unitarie, se $n = 0$

$$A_1 \leftarrow$$

  - clausole goal (o negative, o interrogazioni), se $K = 0$

$$\leftarrow B_1, \dots, B_n$$

    - (un caso particolare) clausole vuote, se anche $n = 0$

□

INTERPRETAZIONE INFORMALE

$A_1 \leftarrow B_1, \dots, B_n$

A_1: testa; $B_1, \dots, B_n$: corpo

(lettura dichiarativa)

- "per ogni istanziazione ground delle variabili, A_1 è vera, se $B_1, \dots, B_n$ sono veri"

(lettura procedurale)

- "per risolvere A_1, risolvi tutti i B_i, $i = 1, \dots, n$"
- "il corpo della procedura A_1 è composto dalle chiamate di procedura $B_1, \dots, B_n$"

- la lettura procedurale è caratteristica delle clausole definite (programmazione logica) e perde significato se si pensa al caso generale, con più di una conclusione.

$\leftarrow B_1, \dots, B_n$

$(\forall Y_1) \dots (\forall Y_k)\ (\neg B_1 \vee \dots \vee \neg B_n)$

$\neg (\exists Y_1) \dots (\exists Y_k)\ (B_1 \wedge \dots \wedge B_n)$

(lettura procedurale)

"risolvi tutti i B_i, $i = 1, \dots, n$"

"chiama le procedure $B_1, \dots, B_n$"

- è la negazione della formula $(\exists Y_1) \dots (\exists Y_k)(B_1 \wedge \dots \wedge B_n)$
- la clausola vuota è la contraddizione dal punto di vista dichiarativo, e la terminazione dal punto di vista procedurale.

PROGRAMMA LOGICO

- un programma logico è un insieme finito non-vuoto di clausole definite

- ed ogni programma logico P è associato un preciso linguaggio del primo ordine L_P : quello definito dai costanti, funzioni e predicati presenti nelle clausole di P
- quando si parla di goal per un programma P, si intendono i goal nel linguaggio L_P

un esempio: P

```
sort(X, Y) ← sorted(Y), perm(X, Y)
sorted(nil) ←
sorted(cons(X, nil)) ←
sorted(cons(X, cons(Y, Z))) ← le(X, Y), sorted(cons(Y, Z))
perm(nil, nil) ←
perm(cons(X, Y), cons(U, V)) ←
                    delete(U, cons(X, Y), Z), perm(Z, V)
delete(X, cons(X, Y), Y) ←
delete(X, cons(Y, Z), cons(Y, W)) ← delete(X, Z, W)
le(0, X) ←
le(f(X), f(Y)) ← le(X, Y)
```

un goal per P

```
← sort(cons(f(0), cons(0, cons(f(f(0)), nil))), X)
```

PROGRAMMAZIONE LOGICA VS DEDUZIONE AUTOMATICA

- la programmazione logica è un caso particolare di theorem proving (con la risoluzione) al prim'ordine
  - utilizza le forme a clausole
  - riconduce il problema di determinare se una certa formula è conseguenza logica di un insieme di assiomi ad un problema di insoddisfacibilità.
  - utilizza il metodo di risoluzione di Robinson (metodo di Herbrand)

- le differenze (specificità)
  - gli assiomi sono clausole Horn definite
  - le formule che siamo interessati a dimostrare hanno tutte la forma

$$(\exists X_1)\ldots(\exists X_k)(B_1 \wedge \ldots \wedge B_n)$$

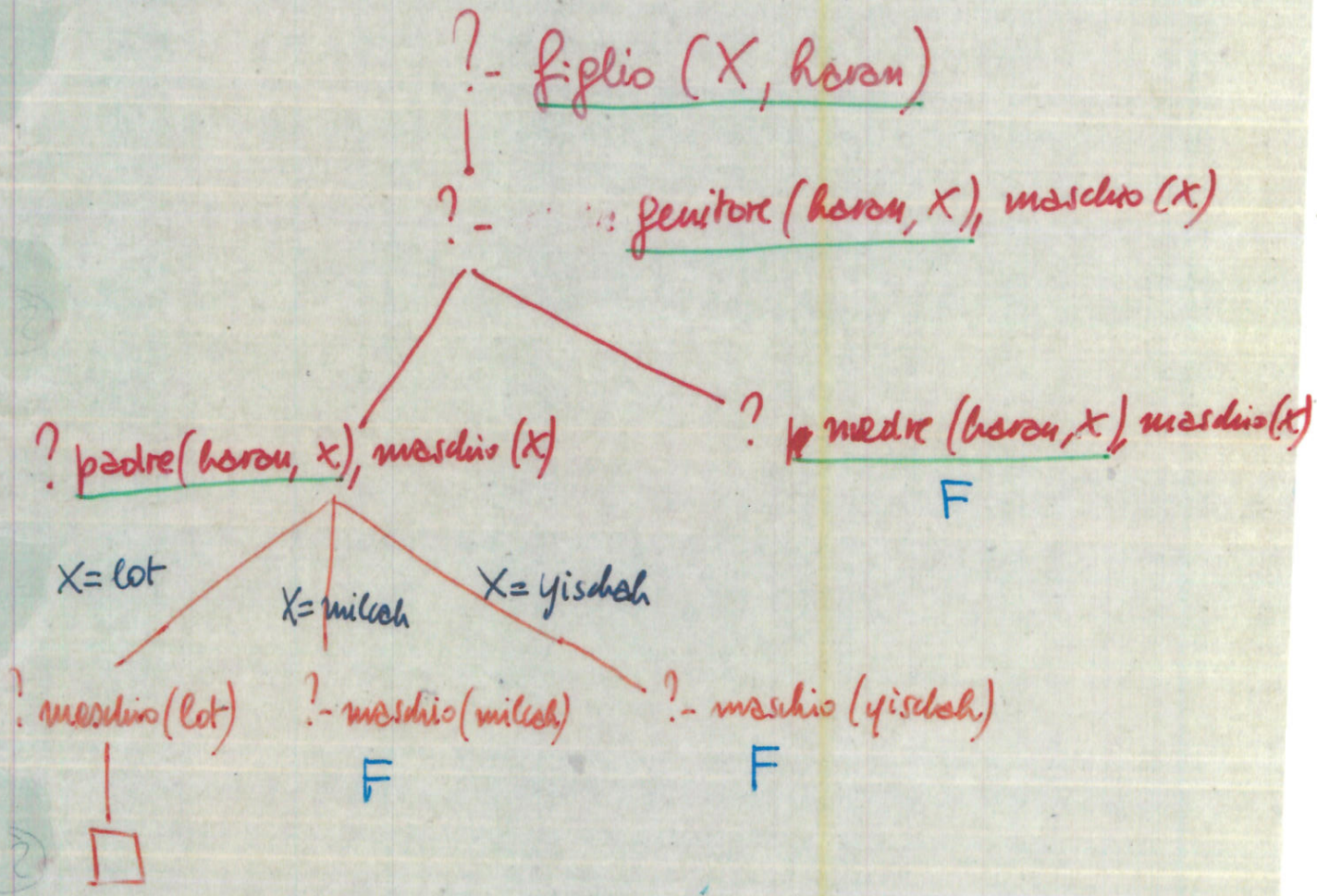
che negate e trasformate in clausole diventano

$$\leftarrow B_1, \ldots, B_n$$

- le conseguenze
  - esistenza di una strategia completa efficiente (≃ lineare ∧ input), che si può vedere come derivare in termini di risoluzione (nondeterministica) di goals
  - quando (se) troviamo $\square$, non solo abbiamo dimostrato quanto volevamo, ma otteniamo una n-upla di termini, per cui vale la formula

dimostrare → calcolare

```
padre (haran, lot).
padre (haran, milcah).
padre (haran, yiscah).
maschio (lot)
genitore (X, Y) :- padre (X, Y).
genitore (X, Y) :- madre (X, Y).
figlio (X, Y) :- genitore (Y, X), maschio (X).
```



INTERPRETAZIONE PROCEDURALE DELLE CLAUSOLE HORN

- clausole definite (regole e fatti) ⟹ dichiarazioni di procedura
- goal ⟹ chiamate di procedura
- goal vuoto ⟹ comando di terminazione

A :- B, C, D

A: Testa della procedura — nome e struttura dei parametri

B, C, D: Corpo della procedura — insieme di chiamate di procedura

- la regola di inferenza (SLD-resolution) è simile alla semantica operazionale delle procedure
- rimpiazza la chiamata con una copia del corpo, dopo aver legato i parametri
- due differenze fondamentali
  - il passaggio di parametri (e ritorno dei risultati) è più complesso (unificazione e variabili logiche)
  - il rimpiazzamento può in generale essere eseguito in modo non deterministico

UNIFICAZIONE E VARIABILI LOGICHE

- l'unificazione di una chiamata di procedura con la testa delle procedure è
  - passaggio di parametri, quando si legano le variabili della clausola (testa), cioè i parametri formali.
  - ritorno di valori, quando si legano le variabili del goal (chiamata)
- l'unificazione fa sì che il ruolo di un argomento non sia fissato (o di ingresso o di uscita)
- i parametri di un goal possono essere termini (strutture dati complete) contenenti variabili (strutture dati parziali)
  - la procedura riceve dall'esterno un valore parziale e può calcolarne una ulteriore approssimazione
- una variabile del goal può venire legata ad un valore parziale
  - è il modo di calcolare la sostituzione di risposta per approssimazioni successive
- i linguaggi logici puri non hanno operazioni primitive
- tutto il potere di calcolo è basato sull'unificazione di strutture dati (parziali)

STRUTTURE DATI

- Il meccanismo dei termini permette di definire in modo naturale tipi di dato per prodotto cartesiano (record) e ricorsione
  - nel linguaggio puro non è previsto alcun vero meccanismo di tipizzazione, macchina nuova è banale (linguaggio del primo ordine)

- due tipi ricorsivi: i naturali e le liste lineari

- naturali
  - costante: 0
  - funzioni: s : nat → nat
  - alcuni dati: 0, s(s(0)), s(s(s(s(s(0))))))
- liste lineari
  - costante: nil
  - funzioni: cons : T × lista(T) → lista(T)
  - alcuni dati: nil, cons(0, nil), cons(s(0), cons(s(0), cons(0, nil)))

- un tipo "record": la data
  - funzione: mkdata : giorno × mese × anno → data
  - alcuni dati: mkdata(15, 3, 90), mkdata(30, 2, 91)

VARIABILI LOGICHE

- I valori di ogni tipo contengono anche i valori parziali

  s(X)
  cons(Y, cons(Z, W))
  mkdata(30, Y, 89)

- Rappresentano insiemi di valori (possibilmente ∞)

- Quando si usano strutture dati parziali nella testa delle dichiarazione di procedura, l'unificazione svolge la funzione di selezione dei componenti della struttura dati o di costruzione di strutture parziali:

  f(cons(X,Y), Z, cons(X,W)) ← f(Y,Z,W)

  ← f(cons(0, cons(0, nil)), W1, Z1)

  Input: {X ← 0, Y ← cons(0, nil)}
  output: {W1 ← Z, Z1 ← cons(0, W)}

  selezione dei componenti delle liste
  costruzione di un output parziale

- strutture dati parziali nel goal corrispondono a passare valori da verranno eventualmente raffinati dalla procedura

IL COMPORTAMENTO DELLE PROCEDURE

$+(0, X, X) \leftarrow$

$+(S(X), Y, S(Z)) \leftarrow +(X, Y, Z)$

- può essere usato in molti modi diversi
  - per calcolare la somma di due numeri
    $\leftarrow +(S(0), S(0), X)$ 1 soluzione
  - per calcolare la differenza
    $\leftarrow +(S(0), Y, S(S(0)))$ 1 o nessuna soluzione
  - per calcolare le coppie che hanno un dato numero come somma
    $\leftarrow +(X, Y, S(S(S(0)))))$ n soluzioni
  - per calcolare le triple (X, Y, Z) tali che $X+Y=Z$
    $\leftarrow +(X, Y, Z)$ ∞ soluzioni

CLASSI DI PROGRAMMI TIPICI

- data base deduttivi
- "algoritmi" di manipolazione di strutture dati ricorsive (à la LISP)
- Rappresentazione delle conoscenze
- Specifiche di algoritmi e prototipazione rapida

DATA-BASE DEDUTTIVI

- un insieme di clausole unitarie ground può essere visto come un data base relazionale
- valutare un goal è come valutare una interrogazione

padre(c, j) ←
padre(p, m) ←
padre(m, c) ←
madre(mm, j) ←
madre(b, mm) ←
madre(jj, c) ←

- aggiungendo delle clausole definite non unitarie otteniamo un data base deduttivo

genitore(X, Y) ← madre(X, Y)
genitore(X, Y) ← padre(X, Y)
} viste

antenato(X, Y) ← genitore(X, Y)
antenato(X, Y) ← genitore(Z, Y), antenato(X, Z)

query ricorsive?

PROGRAMMAZIONE SUI TIPI DI DATO RICORSIVI

- come nei linguaggi funzionali, a parte la notazione

```
append (nil, X, X) ←
append (cons(X,Y), Z, cons(X,W)) ← append(Y, Z, W)

nrev (nil, nil) ←
nrev (cons(X,Y), Z) ← nrev(Y, W),
                      append(W, cons(X, nil), Z)
```

- si ottiene di più
- un'unica definizione può essere utilizzata per calcolare una qualunque proiezione della relazione

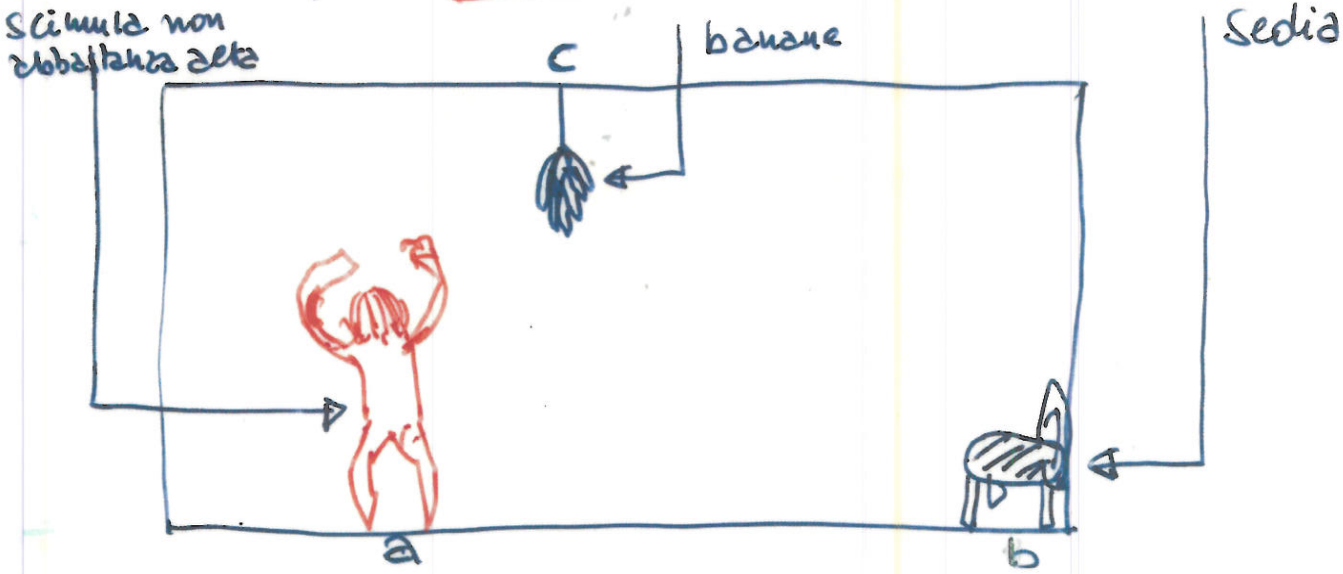
```
← append (X, Y, cons(a, cons(b, nil)))
← append (X, Y, Z)
← append (cons(a, X), cons(b, Y), Z)
```

etc.

RAPPRESENTAZIONE DI CONOSCENZA

- descrizione del problema invece che descrizione dell'algoritmo di soluzione

- un esempio: "la scimmia e le banane"



stato iniziale s_0

trasformatori di stato (azioni della scimmia)

cammina : posizione × posizione × stato → stato
spinge : posizione × posizione × stato → stato
sale : stato → stato

i predicati che descrivono il problema

scimmia : posizione × stato
sedia : posizione × stato
sulla-sedia : posizione × stato
prese! : stato

scimmia (a, S_0) ←
sedia (b, S_0) ←

prese! (S) ← sulle-scolia (c, S)
sulle_sedia (X, sale (S)) ← scimmia (X, S), sedia (X, S)

scimmia (Y, cammina (X, Y, S)) ← scimia (X, S)
scimmia (Y, spinge (X, Y, S)) ← scimmia (X, S), sedia (X, S)

sedia (Y, spinge (X, Y, S)) ← scimmia (X, S), sedia (X, S)
sedia (X, cammina (Y, Z, S)) ← sedia (X, S)

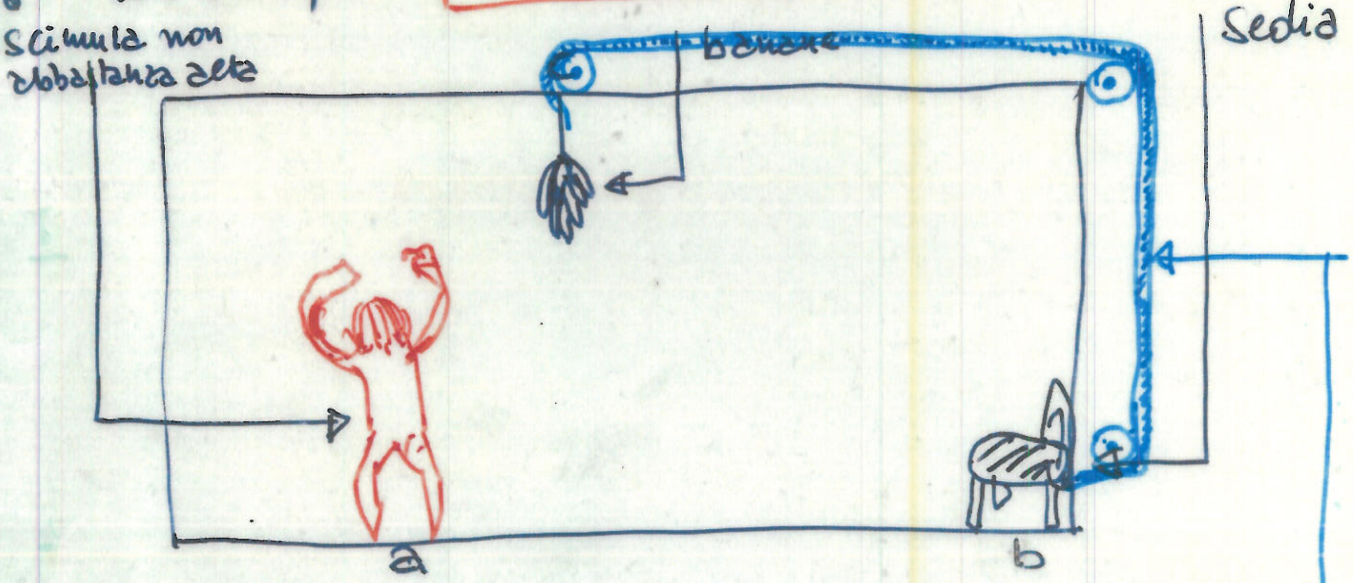
S ← sale (spinge (b, c, cammina (a, b, S_0)))

- perché rappresentare la conoscenza (programmazione dichiarativa) invece che l'algoritmo (programmazione procedurale)?
  - conoscenza incompleta, flessibilità, manutenzione

RAPPRESENTAZIONE DI CONOSCENZA

- descrizione del problema invece che descrizione dell'algoritmo di soluzione

- un esempio: "la scimmia e le banane" : versione vera!



stato iniziale S_0

un vincolo prima trascurato

trasformatori di stato (azioni della scimmia)

cammina : posizione × posizione × stato → stato
spinge : posizione × posizione × stato → stato
sale : stato → stato

i predicati che descrivono il problema

scimmia : posizione × stato
sedia : posizione × stato
sulla-sedia : posizione × stato
prese! : stato

SPECIFICHE E PROTOTIPAZIONE RAPIDA

- i programmi logici puri, letti in modo dichiarativo (logico), sono delle ottime specifiche di programmi

- specifica di "sort", compatta, comprensibile ed eseguibile

```
sort(X, Y) ← perm(X, Y), sorted(Y)

sorted(nil) ←
sorted(cons(X, nil)) ←
sorted(cons(X, cons(Y, Z))) ← X ≤ Y, sorted(cons(Y, Z))

perm(nil, nil) ←
perm(X, cons(Y, Z)) ← append(X1, cons(Y, X2), X),
                      append(X1, X2, X3),
                      perm(X3, Z)
```

- con una regola di selezione rightmost, genera le liste ordinate (se ci riesce, cioè se ha una regola di ricerca fair) e poi controlla se sono permutazioni di quella fornita.

- con una regola leftmost, genera le permutazioni e poi controlla se sono ordinate.

- con una regola a coroutine (pseudo-parallelismo AND) controlla l'ordinamento a mano a mano che vengono generati nuovi elementi nelle permutazioni

ALGORITMO = LOGICA + CONTROLLO