Linguaggi di Programmazione 2020-2021

Prolog e Programmazione Logica IV

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Altri elementi di Prolog

- Predicati meta-logici
- Ispezione di termini
- Predicati di ordine superiore

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

PREDICATI META-LOGICI

Predicati meta-logici: motivazioni

- Considerate questo predicato:

```
celsius_fahrenheit(C, F) :- C is 5/9 * (F - 32).
```

- Il predicato non è *invertibile* (provare per credere) e l'introduzione della seconda clausola

```
celsius_fahrenheit(C, F) :- F is (9/5 * C) + 32.
```

  non aiuta, dato che il sistema si blocca (con un errore) sulla prima clausola

- Il problema è che dobbiamo decidere quale è l'*input* e quale è l'*output* del nostro calcolo
- Per risolvere questo problema abbiamo bisogno di predicati **meta-logici**

Predicati meta-logici

- Alcuni predicati quindi (ad esempio **`minimum`**, **`celsius_fahrenheit`**) non hanno la tipica *invertibilità* dei risultati di varie queries

- La ragione di questo effetto sta nell'uso che abbiamo fatto di vari predicati aritmetici nel corpo dei predicati (**`>`**, **`<`**, **`=<`**, **`is`**, etc)

- Ovvero, per poter usare i predicati aritmetici che usano direttamente l'hardware abbiamo sacrificato la semantica dei nostri programmi

Predicati meta-logici

- I predicati meta-logici principali trattano le variabili come oggetti del linguaggio e ci permettono di riscrivere molti programmi che usano i predicati aritmetici di sistema come predicati dalla semantica "*corretta*" ed dal comportamento invertibile

- I predicati meta-logici più importanti sono
  `var(X)`: vero se **`X`** è una variabile logica
  `nonvar(X)`: l'opposto di **`var(X)`**

- **Esempi**

```
?- var(foo).
false
?- var(X).
tre
?- nonvar(42).
true
```

Predicati meta-logici: esempio

- Il risultato per **`celsius_fahrenheit`** è il seguente

```
celsius_fahrenheit(C, F) :-
    var(C), nonvar(F), C is 5/9 * (F - 32).
celsius_fahrenheit(C, F) :-
    var(F), nonvar(C), F is (9/5 * C) + 32.
```

- L'uso di **`var(X)`** ci permette di decidere quale clausola utilizzare

- Quindi l'uso di questi predicati ci permette di scrivere dei programmi efficienti ed allo stesso tempo semanticamente "corretti"

  - Domanda: cosa succede se C ed F sono entrambe variabili? Ovvero se la query è del tipo **`?- celsius_fahrenheit(X, Y).`** ?
  - Possiamo usare dei cuts per scrivere un programma ancora più completo?

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

ISPEZIONE DI TERMINI

Ispezione di termini

- Finora abbiamo visto come si usano dei termini per rappresentare diverse strutture dati in Prolog

- In particolare abbiamo anche intuito che esistono termini **atomici** (corrispondenti alle costanti in un linguaggio logico del primo ordine) e termini **composti** (funzioni e predicati)

- In Prolog abbiamo a disposizione i predicati

  `atomic(X)` : vero se `X` è un numero od una costante

  `compound(X)` : vero se non `atomic(X)`

Ispezione di termini

- Dato un termine **`Term`** abbiamo tre predicati che ci tornano utili per manipolarlo

- **`functor(Term, F, Arity)`**
  vero se **`Term`** è un termine, con **`Arity`** argomenti, il cui **funtore** (simbolo di *funzione* o di *predicato*) è **`F`**

- **`arg(N, Term, Arg)`**
  vero se l'**`N`**-esimo argomento di **`Term`** è **`Arg`**

- **`Term =.. L`**
  questo predicato, **`=..`**,viene chiamato (per motivi storici) **univ**; è vero quando **`L`** è una lista il cui primo elemento è il *funtore* di **`Term`** ed i rimanenti elementi sono i suoi argomenti

Ispezione di termini

- **Esempi**

```
?- functor(foo(24), foo, 1).
true

?- functor(node(x, _, [], []), F, 4).
F = node

?- functor(Term, bar, 2).
Term = bar(_0,_1)
```

Ispezione di termini

- **Esempi**

```
?- arg(3, node(x, _, [], []), X).
X = []

?- arg(1, father(X, lot), haran).
X = haran
```

Ispezione di termini

- **Esempi**

```
?- father(haran, lot) =.. Ts.
Ts = [father, haran, lot]

?- father(X, lot) =.. [father, haran, lot].
X = haran
```

DIPARTIMENTO DI INFORMATICA SISTEMISTICA E COMUNICAZIONE

PROGRAMMAZIONE DI ORDINE SUPERIORE

Programmazione di ordine superiore

- Quando si formula una domanda per il sistema Prolog, ci si aspetta una risposta che è un'istanza (individuale) derivabile dalla knowledge base

- Il meccanismo di backtracking ci permette di estrarre tutte le istanze che possono essere derivate, una alla volta

- ***Cosa succede se vogliamo come risultato l'insieme di tutte le istanze (soluzioni) che soddisfano una certa query?***

Programmazione di ordine superiore

- Questa richiesta non è una richiesta formulabile direttamente al primo ordine (ovvero non è una richiesta formulabile in un linguaggio logico del primo ordine)

- La richiesta è al secondo ordine, dato che richiede un insieme di elementi che soddisfano una certa proprietà

- Il Prolog mette a disposizione dell'utente una serie di **predicati su insiemi** che *estendono il modello computazionale* del linguaggio di base

Predicati su insiemi

- I predicati su insiemi più importanti sono tre

- **`findall(Template, Goal, Set)`**:
  - Vero se **`Set`** contiene tutte le istanze di **`Template`** che soddisfano **`Goal`**
  - Le istanze di **`Template`** vengono ottenute mediante backtracking
- **`bagof(Template, Goal, Bag)`**:
  - Vero se **`Bag`** contiene tutte le alternative di **`Template`** che soddifano **`Goal`**
  - Le alternative vengono costruite facendo backtracking solo se vi sono delle variabili libere in **`Goal`** che non appaiono in **`Template`**
  - È possibile dichiarare quali variabili non vanno considerate libere al fine del backtracking grazie alla sintassi **`Var^G`** come **`Goal`**
    - In questo caso **`Var`** viene pensata come una variabile esistenziale
- **`setof(Template, Goal, Set)`**:
  - Si comporta come **`bagof`**, ma **`Set`** non contiene soluzioni duplicate

Esempi

- Assumiamo di avere a disposizione il solito database di alberi genealogici

```
?- findall(C, father(X, C), Kids).

Kids = [abraham, nachor, haran, isaac, lot, milcah, yiscah]
```

Esempi

- Un esempio con **bagof**

```
?- bagof(C, father(X, C), Kids).

X = terach
KIDS = [abraham, haran, nachor];

X = haran
KIDS = [lot, yiscah, milcah];

X = abraham
KIDS = [isaac];

false
```

Esempi

- **bagof** con variabile esistenziale

```
?- bagof(C, X^father(X, C), Kids).

Kids = [abraham, haran, lot, yiscah, nachor, isaac, milcah];

false
```

Predicati di ordine superiore e meta variabili

- Il Prolog mette a disposizione dell'utente anche altri predicati di ordine superiore
- Buona parte di questi predicati funziona grazie al meccanismo delle meta-variabili, ovvero variabili interpretabili come goals

- Un esempio tipico è il predicato **`chiama`** che si può pensare essere definito come

```
chiama(G) :- G.
```

  - Il predicato standard SWI-Prolog si chiama **`call`**.

Predicati di ordine superiore e meta variabili

- Grazie alle meta variabili possiamo definire il predicato **applica** che valuta una query composta da un funtore e da una lista di argomenti

```
applica(P, Argomenti) :-
      P =.. PL, append(PL, Argomenti, GL), Goal =.. GL, call(Goal).
```

- **Esempio**

```
?- applica(father, [X, C]).
X = terach
C = abraham;
X = terach
C = nachor;
false

?- applica(father(terach), [C]).
C = abraham;
C = nachor;
false
```

DIPARTIMENTO DI INFORMATICA SISTEMISTICA E COMUNICAZIONE

MANIPOLAZIONE DELLA BASE DATI

Cose da fare con molta attenzione

- Un programma Prolog è costituito da una *base di dati* (o *knowledge base*) che contiene *fatti* e *regole*.
- Il Prolog però mette a disposizione anche altri predicati che servono a manipolare direttamente la base di dati. Ovviamente, questi predicati vanno usati con molta attenzione, dato che modificano dinamicamente lo stato del programma.
- I predicati che servono a manipolare direttamente la base di dati sono
  - **`listing`**
  - **`assert`**, **`asserta`**, **`assertz`**
  - **`retract`**
  - **`abolish`**

Manipolazione base dati

- Immaginiamo di partire con una knowledge base vuota. Se si lancia il comando:

```
?- listing.
Yes
```

- La risposta sarà semplicemente `Yes`; il “listing” é ovviamente vuoto, ovvero la base dati corrente è vuota.

Manipolazione base dati

- Consideriamo invece il comand seguente.

```
?- assert(happy(maya)).
true
```

- Il comando ha successo (assert ha sempre successo). Tuttavia l'importanza di questo comando non è il suo successo, ma il suo effetto collaterale sullo stato del database. Se proseguiamo con la query listing, ottieniamo il seguente effetto:

```
?- listing.
happy(maya).
true
```

- Ovvero, la base dati non è più vuota: ora contiene il fatto che abbiamo asserito (con `assert`).

Manipolazione base dati

- Eseguiamo quattro nuove assert:

```
?- assert(happy(vincent)).
true

?- assert(happy(marcellus)).
true

?- assert(happy(butch)).
true

?- assert(happy(vincent)).
true
```

Manipolazione base dati

- A questo punto chiediamo il listing:

```
?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
true
```

- Tutti I fatti che abbiamo asserito si trovano nella knowledge base. Notate che `happy(vincent)` si trova due volte nella knowledge base. Ciò non stupisce dato che lo abbiamo asserito due volte.

Manipolazione base dati

- Finora abbiamo solo asserito (aggiunto) dei *fatti* nella base di dati, ma possiamo anche asserire delle regole. Ad esempio, possiamo asserire la regola - molto ottimista - che chiunque è felice è ingenuo (*naïve*). Ovvero, vorremmo asserire

```
naive(X) :- happy(X).
```

- Per far ciò usiamo il comando:

```
?- assert( (naive(X) :- happy(X)) ).
true
```

- Notate la sintassi di questo comando: la regola che stiamo asserendo è racchiusa tra parentesi. Se ora chiediamo il listato della knowledge base otteniamo:

```
?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
naive(A) :-
      happy(A).
true
```

Manipolazione base dati

- Ora che sappiamo come possiamo asserire fatti e regole, ovvero nuove informazioni, nella base di dati, possiamo chiederci come possiamo fare l'operazione inversa. Ovvero come possiamo rimuovere dalla knowledge base fatti e regole che non ci interessano più. Il predicato inverso di `assert` è `retract`. Ad esempio, possiamo dare direttamente il comando seguente:

```
?- retract(happy(marcellus)).
true
```

e chiedere il listato del programma; ciò che otteniamo è:

```
?- listing.
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).
naive(A) :-
   happy(A).
true
```

Ovvero, il fatto `happy(marcellus)` è stato rimosso.

Manipolazione base dati

- Supponiamo di procedere oltre lanciando in comando seguente:

```
?- retract(happy(vincent)).
true
```

- Il listato che si ottiene è:

```
?- listing.
happy(mia).
happy(butch).
happy(vincent).
naive(A) :-
   happy(A).
true
```

- Notate che *solo la prima occorrenza* di `happy(vincent)` è stata rimossa.

Manipolazione base dati

- Per rimuovere tutte le nostre asserzioni possiamo usare una variabile.

```
?- retract(happy(X)).
X = mia;
X = butch;
X = vincent;
false
```

- La richiesta di listare il programma risulta quindi in:

```
?- listing.
naive(A) :-
   happy(A)
true
```

Manipolazione base dati

- Per avere più controllo su dove vengono aggiunti fatti e regole possiamo usare le due varianti di `assert`, ovvero:
  - **`assertz`**
    Inserisce l'asserzione alla fine della knowledge base.
  - **`asserta`**
    inserisce l'asserzione all'inizio della knowledge base.
- Ad esempio, se partiamo con una base di dati vuota, e diamo il seguente comando:

```
?- assert(p(b)), assertz(p(c)), asserta(p(a)).
true
```

- Il risultato del comando listing sarà:

```
?- listing.
p(a).
p(b).
p(c).
true
```

Manipolazione base dati

- La manipolazione del database Prolog è una cosa molto utile

- Ad esempio, può essere usata per memorizzare i risultati intermedi di varie computazioni, in modo da non dover rifare delle queries dispendiose in futuro: semplicemente si ricerca direttamente il fatto appena asserito

- Questa tecnica si chiama *memoization* o *caching* (in Inglese)

Manipolazione base dati

- Ecco un esempio. Creiamo una tavola di addizioni manipolando la knowledge base. Consideriamo il programma seguente:

```
addition_table(A) :-
        member(B, A),
        member(C, A),
        D is B + C,
        assert(sum(B, C, D)),
        fail.
```

In questo esempio `member/2` è il predicato standard che controlla l'appartenenza di un elemento in una lista

Manipolazione base dati

- Consideriamo il seguente esempio

```
?- addition_table([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
false
```

- La risposta è *false*; ma non è la risposta che ci interessa, bensì l'effetto (collaterale) che l'interrogazione ha sulla knowledge base
- Se ora chiediamo un listato della base dati otteremo

```
?- listing(sum).
sum(0, 0, 0).
sum(0, 1, 1).
sum(0, 2, 2).
sum(0, 3, 3).
…
sum(9, 9, 18).
true
```

Manipolazione base dati

- Domanda: come possiamo rimuovere tutti questi fatti quando non li vogliamo più nel data base? È vero che potremmo semplicemente dare il comando:

```
?- retract(sum(X, Y, Z)).
```

- Ma in questo casi il Prolog ci chiederebbe se vogliamo rimuovere tutti i fatti uno per uno!
- In realtà c'è un modo più semplice e diretto: usiamo il comando:

```
?- retract(sum(_, _, _)), fail.
No
```

- Ancora una volta, lo scopo del `fail` è di forzare il backtracking. Il Prolog rimuove il primo fatto con funtore sum dalla base di dati e poi fallisce. Quindi fa bactrack e rimuove il fatto successivo e così via. Alla fine, dopo aver rimosso tutti i fatti con funtore `sum`, la query fallirà completamente ed il Prolog risponderà (correttamente) con un No. Ma anche in questo caso a noi interessa unicamente l'effetto - *collaterale* - sulla knowledge base.

Sommario

- A conclusione, si ripetono gli argomenti trattati in questa serie di slides.
  - Predicati meta-logici
  - Ispezione di termini
  - Predicati di ordine superiore