UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

BCC35A - Linguagens de Programação

Prof. Dr. Rodrigo Hübner

Aula 10: Linguagens de programação lógicas

Introdução

- Na programação **lógica** ou **declarativa** é utilizado o processo de **inferência lógica**
- É declarativa ao invés de procedural, pois apenas a **especificação do resultado** é dada e não um processo detalhado de **como obter o resultado**
- Vamos relembrar o cálculo de predicados

Cálculo de predicados

- **Proposição** pode ser verdadeira ou falsa
- Objetos em proposições são representados por **contantes** e **variáveis**
- **Termo composto**:
  1. **functor**, o símbolo de função que nomeia a relação
  2. lista ordenada de parâmetros

     Ex.: `man(jake)`, `like(bob, steak)`

Cálculo de predicados

- Proposição pode ser declarada de dois modos:
  - **Fato**: a proposição é definida como verdadeira
  - **Consulta**: a verdade da proposição precisa ser determinada
- No cálculo de predicados existem muitas formas de expressar a mesma coisa, o que é ruim para implementação
- simplificamos qualquer proposição pela **forma clausal**:
  - `B1 ∪ B2 ∪ · · · ∪ Bn ⊂ A1 ∩ A2 ∩ · · · ∩ Am`
  - se todos os `A`'s forem verdadeiros, pelo menos um `B` é verdadeiro

Cálculo de predicados

- Outros exemplos:
  - `likes(bob, trout) ⊂ likes(bob, fish) ∩ fish(trout)`
  - `father(louis, al) ∪ father(louis, violet) ⊂ father(al, bob) ∩ mother(violet, bob) ∩ grandfather(louis, bob)`
  - **Como são lidas em português?**

Demonstração de teoremas

- `older(joanne, jake) ⊂ mother(joanne, jake)`
- `wiser(joanne, jake) ⊂ older(joanne, jake)`
- A partir destas proposições, a seguinte proposição pode ser construída usando resolução:
- `wiser(joanne, jake) ⊂ mother(joanne, jake)`

Demonstração de teoremas

- A resolução é mais complexa do que parece nos exemplos...
- Determinar valores que permitam o casamento é chamado de **unificação**
- Atribuição temporária de valores as variáveis que permite a unificação é chamada **instanciação**
- Se o casamento falhar após a instanciação, pode ser necessário retroceder (***backtrack***).
- As proposições iniciais são chamadas de **hipóteses**
- Negação do teorema é a **meta** $\rightarrow$ demonstrar é achar **inconsistência**!

Demonstração de teoremas

- Para simplificar a resolução, as proposições devem estar em uma forma clausal restrita, chamada de **cláusulas de Horn**
  - No máximo **uma proposição atômica** do lado **esquerdo**
  - Lado esquerdo $\rightarrow$ **cabeça**; Lado direito $\rightarrow$ **cauda**
  - **Quase todas** as proposições podem ser escritas como cláusulas de Horn

A linguagem de programação `Prolog`

- Todas as sentenças em `Prolog` são construídas com **termos**:
  - **Constante** (Átomo ou Inteiro)
    - Uma string de letras, dígitos e underscores, começando com uma letra minúscula
  - **Variável**
    - Qualquer string de letras, dígitos e underscores, começando com uma letra maiúscula
  - **Estrutura**
    - `functor(lista de parâmetros)`

Os elementos básicos de `Prolog`

- **Fatos** (cláusulas de Horn sem cabeça):

```
mulher(maria).
homem(joao).
mulher(paula).
homem(jose).
pai(joao, jose).
pai(joao, maria).
mae(paula, jose).
mae(paula, maria).
```

Os elementos básicos de `Prolog`

- **Regras** (cláusulas de Horn com cabeça):
  - Forma geral: `consequente :- antecedentes`
  - Exemplo:

```
pais(X, Y) :- mae(X, Y).
pais(X, Y) :- pai(X, Y).
avos(X, Z) :- pais(X, Y), pais(Y, Z).
irmaos(X, Y) :- mae(M, X), mae(M, Y), pai(P, X), pai(P, Y).
```

- Para todo `X` e `Y`, `X` é irmão de `Y` se existem `M` e `P` tal que `M` é mãe de `X` e `Y` e `P` é pai de `X` e `Y` .

Os elementos básicos de `Prolog`

- Casamento de preposição:
  - Meta: `homem(bob).`
  - Se a base de dados contém o fato `homem(bob)` a prova é trivial
  - Se a base de dados contém:

```
pai(bob).
homem(X) :- pai(X).
```

  - o `Prolog` deve usar estas duas declarações para inferir que a meta é **verdadeira**

O processo de inferência do `Prolog`

- **Resolução de cima para baixo** (***top-down***), encadeamento para trás
  - Inicia com a meta e tenta encontrar uma sequência que leva a um conjunto de fatos na base de dados
- Quando uma meta tem mais que uma submeta, o `Prolog` usa **busca em profundidade**
- Com uma meta composta, se a prova de uma submeta falha, é necessário considerar as submetas anteriores (**retroceder** ou ***backtrack***)

O processo de inferência do `Prolog`

- Existem 4 eventos de rastreamento (*trace*):
  - `call` início da tentativa de satisfazer uma meta
  - `exit` quando uma meta foi satisfeita
  - `redo` quando ocorre retrocesso
  - `fail` quando uma meta falha

Inferência do `Prolog` (exemplo *track*)

```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

likes(jake, X), likes(darcie, X).
   Call: (7) likes(jake, _G367) ?
   Exit: (7) likes(jake, chocolate) ?
   Call: (7) likes(darcie, chocolate) ?
   Fail: (7) likes(darcie, chocolate) ?
   Redo: (7) likes(jake, _G367) ?
   Exit: (7) likes(jake, apricots) ?
   Call: (7) likes(darcie, apricots) ?
   Exit: (7) likes(darcie, apricots) ?
X = apricots.
```

Aritmética

- Operador `is`:
  - `A is B / 17 + C`
- Exemplo:

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed), time(X, Time), Y is Speed * Time.
```

A "negação" de `Prolog`:

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- parent(M, X), parent(M, Y).
```

- Se realizar a consulta `sibling(X, Y)` temos `X = Y, Y = jake .`
- Realizamos então:

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- parent(M, X), parent(M, Y), not(Y == X).
?- sibling(X, Y).
X = jake,
Y = shelley .
```

Listas

- Os elementos podem ser átomos, proposições atômicas, ou qualquer outro termo, incluindo outras listas:
  `[maça, laranja, amora, pera]`

```
[]% lista vazia
[ X | Y ] % cabeça X e cauda Y. Usado para construir e desmanchar listas
```

Exemplos: ...

```
% EXEMPLO de "membro"
membro(Element, [Element | _]).
membro(Element, [_ | Tail]) :- membro(Element, Tail).

membro(a, [b, c, d]).
  Call: (6) membro(a, [b, c, d]) ?
  Call: (7) membro(a, [c, d]) ?
  Call: (8) membro(a, [d]) ?
  Call: (9) membro(a, []) ?
  Fail: (9) membro(a, []) ?
  Fail: (8) membro(a, [d]) ?
  Fail: (7) membro(a, [c, d]) ?
  Fail: (6) membro(a, [b, c, d]) ?
false.
[trace] ?- membro(b, [a, b, c]).
  Call: (6) membro(b, [a, b, c]) ?
  Call: (7) membro(b, [b, c]) ?
  Exit: (7) membro(b, [b, c]) ?
  Exit: (6) membro(b, [a, b, c]) ?
true ;
```

```
% EXEMPLO de "concatena"
concatena([], L, L).
concatena([X | L1], L2, [X | L3]) :- concatena(L1, L2, L3).

concatena([bob, jo], [jake, darcie], Family).
   Call: (6) concatena([bob, jo], [jake, darcie], _G380) ?
   Call: (7) concatena([jo], [jake, darcie], _G459) ?
   Call: (8) concatena([], [jake, darcie], _G462) ?
   Exit: (8) concatena([], [jake, darcie], [jake, darcie]) ?
   Exit: (7) concatena([jo], [jake, darcie], [jo, jake, darcie]) ?
   Exit: (6) concatena([bob, jo], [jake, darcie], [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie].
```

```
% EXEMPLO de "reverso"
concatena([], L, L).
concatena([X | L1], L2, [X | L3]) :- concatena(L1, L2, L3).

reverso([], []).
reverso([Head | Tail], List) :- reverso(Tail, Result),
  concatena(Result, [Head], List).

| ?- reverso([ja, ca], Saida).
  1   1   Call: reverso([ja,ca],_27) ?
  2   2   Call: reverso([ca],_96) ?
  3   3   Call: reverso([],_120) ?
  3   3   Exit: reverso([],[]) ?
  4   3   Call: concatena([],[ca],_148) ?
  4   3   Exit: concatena([],[ca],[ca]) ?
  2   2   Exit: reverso([ca],[ca]) ?
  5   2   Call: concatena([ca],[ja],_27) ?
  6   3   Call: concatena([],[ja],_164) ?
  6   3   Exit: concatena([],[ja],[ja]) ?
  5   2   Exit: concatena([ca],[ja],[ca,ja]) ?
  1   1   Exit: reverso([ja,ca],[ca,ja]) ?
Saida = [ca,ja]
```

```
% EXEMPLO de "reverso"
reverso([], []).
reverso([Head | Tail], List) :- reverso(Tail, Result),
  concatena(Result, [Head], List).

reverso([a, b, c], Q).
  Call: (6) reverso([a, b, c], _G376) ?
  Call: (7) reverso([b, c], _G455) ?
  Call: (8) reverso([c], _G455) ?
  Call: (9) reverso([], _G455) ?
  Exit: (9) reverso([], []) ?
  Call: (9) concatena([], [c], _G459) ?
  Exit: (9) concatena([], [c], [c]) ?
  Exit: (8) reverso([c], [c]) ?
  Call: (8) concatena([c], [b], _G462) ?
  Call: (9) concatena([], [b], _G457) ?
  Exit: (9) concatena([], [b], [b]) ?
  Exit: (8) concatena([c], [b], [c, b]) ?
  Exit: (7) reverso([b, c], [c, b]) ?
  Call: (7) concatena([c, b], [a], _G376) ?
  Call: (8) concatena([b], [a], _G463) ?
  Call: (9) concatena([], [a], _G466) ?
  Exit: (9) concatena([], [a], [a]) ?
  Exit: (8) concatena([b], [a], [b, a]) ?
  Exit: (7) concatena([c, b], [a], [c, b, a]) ?
  Exit: (6) reverso([a, b, c], [c, b, a]) ?
Q = [c, b, a].
```

Aplicações da programação lógica

- Sistema de gerenciamento de banco de dados relacional
- Sistemas especialistas
- Processamento de linguagem natural

Próxima aula

- Acompanhamento de trabalho
- Pré-processamento de linguagens