

Programação Declarativa

Paradigmas de Programação Avançados

Salvador Abreu, Universidade de Évora, 2019/20

Técnicas de Programação

Recursão

Números

Listas

~~Meta-programação~~

~~Uso da unificação~~

Acumulador

~~Termos “abertos” (p/ex listas de diferença)~~

Demonstração em sistemas formais

Se temos um sistema **hipotético-dedutivo**, por exemplo uma **álgebra**, podemos especificá-la como:

- Um conjunto de **axiomas**.
 - Regras
 - Verdades absolutas (i.e. não questionáveis)

Dado este, podemos raciocinar sobre ele permite-nos demonstrar **verdades derivadas**, i.e. teoremas.

O objetivo é definir coisas destas e o Prolog vai revelar-se muito útil.

Como exprimir coisas repetitivas

Podemos fazer — à moda “clássica” — um ciclo:

- Inicialização
- Validação de pré-condições
- Enquanto que as condições de continuação se verificarem,
 - Fazer o que há a fazer
 - Reinicializar (avançar)
- Validação de pós-condições

Problemas:

- Estado **muda**, i.e. posso fazer $x = x+1$ ($x_{t+1} = x_t+1$)
- Efeitos secundários escondidos (funções que alterem o estado)

Repetição sem alteração de estado

Especificar:

- Um **passo** (i.e. como é que se avança)
- Um **caso base** (de onde é que se começa, ou onde é que se para)

Importante formular todos os problemas (e sua resolução) nestes termos.

Pode haver mais de um passo ou caso base (funcionam como alternativas).

Numerais










Podemos definir os *numerais de Peano*, assim:

- **z** (zero) é um numeral (de Peano)
- Se **X** for um numeral, então **s(X)** (sucessor de X) também o é

Exprimindo isto em Prolog:

```
num(z).  
num(s(X)) :- num(X).
```

Para mais informações, ver, por exemplo:
https://pt.wikipedia.org/wiki/Axiomas_de_Peano

```
| ?- num(z).   
yes  
| ?- num(s(z)).   
yes  
| ?- num(0).   
no  
| ?- num(10).   
no  
| ?- num(X).   
X = z ?   
X = s(z) ?   
X = s(s(z)) ?   
X = s(s(s(z))) ?   
yes  
| ?-
```

Numerais

Agora, vamos exprimir a **axiomática** sobre os numerais, i.e.

- As relações que queremos que sejam verdade

Também vamos **deduzir** algumas propriedades, escrevendo “teoremas”, sob a forma de **predicados** Prolog.

Numerais: comparações (\leq)

A ideia é dizer que:

- Zero é menor ou igual do que qualquer numeral
- $X+1$ é menor ou igual a $Y+1$ se X for menor ou igual a Y

le(z, X) :- num(X).

le(s(A), s(B)) :- le(A, B).

le == Less than or Equal to

Podemos substituir a primeira regra por:

- Zero é menor ou igual que qualquer coisa

Ou seja:

le(z, _).

le(s(A), s(B)) :- le(A, B).

Numerais: comparações (<)

Como anteriormente, exceto que excluimos o caso (z,z)

- Zero é menor ou igual a qualquer numeral **que seja um sucessor**
- $X+1$ é menor que $Y+1$ se X for menor que Y

`lt(z, s(X)) :- num(X).`

`lt(s(A), s(B)) :- lt(A, B).`



lt == Less Than

Como anteriormente, podemos substituir a primeira cláusula por:

`lt(z, s(_)).`

Para significar que “zero é menor que o sucessor de qualquer coisa”, por oposição a “...sucessor dum numeral”.

Numerais: aritmética

Como fazer contas?

Soma?

- Propriedades algébricas!
- Elemento neutro: $X+0 = X$ e $0+X = X$
- Soma de 1: $(X+1)+Y = (X+Y)+1$
- Não vamos entrar para já com propriedades como a comutatividade ou associatividade

O que fazer?

- Um predicado soma/3, em que
 - Os primeiros 2 argumentos são *as parcelas da soma* e
 - O terceiro argumento é a *soma*
- Traduzimos a definição...



input



output

Numerais: soma

Ficamos com isto

```
soma(z, X, X).  
soma(s(X), Y, s(Z)) :- soma(X, Y, Z).
```

Usamos os termos que compõem os números (**z** e **s/1**)

Nota: estamos a fazer a recursão sobre o primeiro argumento (clausulas diferentes têm valor diferente para o argumento)

Qual o resultado destes goals?

```
soma(s(z), s(s(z)), X).  
soma(s(z), X, s(s(z))).  
soma(X, Y, s(s(s(z)))).  
soma(s(s(z)), Y, s(z)).
```

Desviar a soma...

A definição de **soma/3** é muito mais poderosa do que parece à primeira vista...

Tentemos definir **le/2** em função de **soma/3**:

- $A \leq B$ se for possível dizer que $A+X = B$, para um qualquer numeral X

Trocando por Prolog:

```
le(A, B) :- soma(A, X, B), num(X).
```

Ou simplesmente

```
le(A, B) :- soma(A, _, B).
```

Sabendo que **soma/3** só “engole” numerais...

Mais desvios...

De igual modo, podemos definir **lt/2**, dizendo que:

- $A < B$ se for possível dizer que $A+X = B$, para um qualquer numeral X , desde que X seja superior a zero

Trocando por Prolog:

```
lt(A, B) :- soma(A, X, B), num(X), gt(X, z).
```

Melhor:

```
lt(A, B) :- soma(A, X, B), num(X), X=s(_).
```

Ou ainda:

```
lt(A, B) :- soma(A, s(_), B).
```

Como anteriormente...

Subtração

Como fazer...

- $X = A - B \Leftrightarrow X + B = A$

Portanto... vamos usar a soma para fazer a subtração

sub(A, B, X) :- soma(X, B, A).

| ?- sub(s(s(s(z))), X, Y).

X = s(s(s(z)))

Y = z ? ;

X = s(s(z))

Y = s(z) ? ;

X = s(z)

Y = s(s(z)) ? ;

X = z

Y = s(s(s(z))) ? ;

no

| ?-

Continuando...

Temos definições de propriedades fundamentais (p/ex soma/3) e vamos especificar as outras em função dessas.

Vamos fazer a multiplicação:

- $0 * X = 0$, qualquer que seja X
- Podemos dizer que $A * B = X = (A-1) * B + B$
 - Seja $A=A'+1$, $X = A' * B + B$

Passando para Prolog:

```
mult(z, _, z).  
mult(s(A), B, X) :-  
    mult(A, B, Y),  
    soma(B, Y, X).
```

```
| ?- mult(s(s(z)), s(s(z)), X).
```

```
X = s(s(s(s(z))))
```

```
yes
```

```
| ?- mult(s(s(z)), A, X).
```

```
A = z
```

```
X = z ? ;
```

```
A = s(z)
```

```
X = s(s(z)) ? ;
```

```
A = s(s(z))
```

```
X = s(s(s(s(z)))) ? ;
```

```
A = s(s(s(z)))
```

```
X = s(s(s(s(s(s(z)))))) ?
```

```
(1 ms) yes
```

```
| ?-
```

Divisão

Resta falar da divisão.

Aqui somos puramente declarativos:

- $A/B = X \Leftrightarrow A = XB$

Portanto só temos de dizer

`div(A, B, X) :- mult(X, B, A).`

Isto funciona, mas só no caso de A ser divisível por B (divisão inteira).

Divisão com resto

Digamos que queremos:

- **div(A, B, Q, R)**, que significa $A/B = Q$, com resto R
- equivale a dizer $A = QB + R$
 - Ou seja, existe um $X = Q*B$ tal que $A = X+R$
- Convém que $R < B$

Pronto! Não precisamos dizer mais, basta traduzir cada frase para Prolog:

```
div(A, B, Q, R) :-  
    mult(B, Q, X),  
    soma(X, R, A),  
    lt(R, B).
```

```
| ?- mult(s(s(z)), s(s(z)), X),  
      div(s(X), s(s(z)), Q, R).
```

```
Q = s(s(z))  
R = s(z)  
X = s(s(s(s(z)))) ?
```

O que aconteceria sem o lt/2 no fim?

```
yes  
| ?-
```

Definição auxiliar

Pode dar jeito converter entre numerais de Peano e inteiros naturais, na representação da máquina.

Gostaríamos de ter algo como:

```
| ?- num(z, 0).  
| ?- num(s(s(s(z))), 3).
```

Pode-se definir usando o predicado de **avaliação** de expressões aritméticas, **is/2**.

```
num(z, 0).  
num(s(X), SY) :- num(X, Y), SY is Y+1.
```

Este predicado funciona no sentido numeral \rightarrow inteiro...

Será que também funciona no outro... Porquê? Qual o problema potencial?

Listas

Uma lista é composta por um termo que pode ser

- A lista vazia `[]`, pronunciada “nil”
- Um *par* formado por uma **cabeça** (head) e uma **cauda** (tail).
 - `'.'(CABECA, CAUDA)`
 - `[CABECA | CAUDA]`
 - `[CABECA , .. CAUDA]`

Uma lista é um termo normal, mas com uma sintaxe exterior particular.

Predicado de *tipo* para listas:

```
lista([]).                % caso base
lista([_|L]) :- lista(L). % caso recursivo
```

Lista como conjunto

Pertença a uma lista (“membro”)

```
membro(X, [X|_]).
```

```
membro(X, [_|L]) :- membro(X, L).
```

Agora, vamos estudar alguns predicados para manipular listas.

Prefixo

Uma lista é prefixo de outra:

- A lista vazia é (trivialmente) prefixo de qualquer lista
- Uma lista A é prefixo duma lista B se
 - A cabeça de A for a mesma que a de B, e:
 - A cauda de A for prefixo da cauda de B

Traduzindo para código Prolog:

```
prefixo([], _).  
prefixo([X|A], [X|B]) :- prefixo(A, B).
```

Sufixo

De igual modo, uma lista é sufixo de outra:

- Qualquer lista é (trivialmente) sufixo dela mesma
- Uma lista A é sufixo duma lista B se A for sufixo da cauda de B

Traduzindo para código Prolog:

```
sufixo(A, A).  
sufixo(A, [_|B]) :- sufixo(A, B).
```

Sublista

Uma sublista é um prefixo dum sufixo (ou vice-versa)

```
sublista(S, L) :- prefixo(P, L), sufixo(S, P).
```

Alternativamente, podemos “desenvolver” um dos elementos de definição, neste caso o “sufixo”:

```
sublista(S, L) :- prefixo(S, L)  
sublista(S, [_|L]) :- sublista(S, L).
```

Catenação de listas (“juntar”)

Formulação

- Catenar a lista vazia a uma lista A resulta na lista A
- Catenar uma lista que começa com X e resulta numa lista que também começa com X

Em Prolog:

```
catena([], L, L).
```

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

```
| ?- catena([1,2], [3,4], X).
```

```
X = [1,2,3,4]
```

```
Yes
```

```
| ?- catena(X, [3,4], [1,2,3,4]).
```

```
X = [1,2] ? ;
```

```
no
```

```
| ?-
```


Reutilização de código...

Usando o `catena/3` podemos visitar algumas das coisas que tínhamos definido:

```
prefixo(X, Y) :- catena(X, _, Y).
```

```
sufixo(X, Y) :- catena(_, X, Y).
```

```
membro(X, Y) :- catena(_, [X|_], Y).
```

De igual modo podemos definir predicados úteis, como por exemplo:

```
ultimo(X, Y) :- catena(_, [X], Y).
```

```
adjacente(X, Y, Z) :- catena(_, [X, Y|_], Z).
```

O grande clássico: o naïve-reverse

Trata-se de inverter uma lista.

Dois casos:

1. Se a lista for vazia, também o é a lista inversa
2. Se a lista começar por X, começamos por inverter a cauda da lista, e ao resultado catenamos a lista [X] para obter a lista inversa

Fica assim, o código Prolog:

```
nrev([], []).  
nrev([X|A], B) :-  
    nrev(A, AR),  
    catena(AR, [X], B).    % costuma ser append/3
```

O naïve-reverse é usado como “benchmark” para sistemas Prolog

Inversão de lista com *acumulador*

Ideia: vamos construindo a lista inversa enquanto atravessamos a de entrada. Quando chegarmos ao fim, a que fomos construindo já é a lista invertida...

Código:

```
rev(L, R) :- rev(L, [], R).  
  
rev([], R, R).  
rev([A|B], X, R) :- rev(B, [A|X], R).
```

Modelo:

- Fazer predicado auxiliar (neste caso, rev/3)
- Auxiliar tem mais um parâmetro, dito de acumulação (neste caso, o 2º)
- Auxiliar tem **recursão terminal!**

Contagem de elementos

Podemos usar numerais de Peano:

```
compr([], z).  
compr([_|T], s(X)) :- compr(T, X).
```

Ou então fazemos com os built-ins de aritmética:

```
compr([], 0).  
compr([_|T], X) :- compr(T, Y), X is Y+1.
```

Qual a melhor? Porquê? Quais os problemas?

Remoção dum elemento

Podemos remover um elemento duma lista, especificando o elemento e a lista, “recebendo” a lista da qual foi removido o dito elemento...

```
sel(E, [E|L], L).
```

```
sel(E, [X|L], [X|M]) :- sel(E, L, M).
```

Questão: como remover **todas** as ocorrências??