Programação Declarativa

Licenciatura em Engenharia Informática

2014–2015

Vitor Beires Nogueira

Escola de Ciências e Tecnologia
Universidade de Évora

Código

```
/*
natural_number(N) <-
  N is a natural number
*/

natural_number(0).

natural_number(s(X)) :-
   natural_number(X).
```

Proposição

O programa Números naturais é correcto e completo em relação ao conjunto de goals natural_number($s^i(0)$) para $i \geq 0$.

Menor ou igual

Código

```
/*
less_or_equal(X,Y)
  X and Y are natural numbers
  such that X is less or equal to Y
*/

less_or_equal(0,X) :-
  natural_number(X).

less_or_equal(s(X),s(Y)):-
  less_or_equal(X,Y).
```

Menor

Código

```
/*
   less(X,Y)
   X < Y
*/

less(0,s(X)) :-
        natural_number(X).

less(s(X),s(Y)) :-
        less(X,Y).
```

Menor (ou igual): considerações

O programa para verificar se um número natural é menor (ou igual) a outro não é eficiente.

Definição

Definimos uma *condição de tipo* como sendo a invocação de um predicado que define um tipo.

Código

```
/*
plus(X,Y,Z)
  X, Y and Z are natural numbers
  such that Z is the sum of X and Y
*/

plus(0,X,X) :-
  natural_number(X).

plus(s(X),Y,s(Z)):-
  plus(X,Y,Z).
```

- Programas em lógica que correspondem a funções de n argumentos, definem relações de aridade $n + 1$.
- Para computar o valor da função é efectuada uma *query* com os n argumentos instanciados e o argumento relativo ao valor da função não instanciado.

Exemplo

Qual o resultado de

1. `plus(s(0),s(s(0)),X)?`
2. `plus(s(0),X,s(s(0)))?`
3. `plus(X,Y,s(s(s(0))))?`

Código

```
/*
times(X,Y,Z)
  X, Y and Z are natural numbers
  such that Z is the product of X and Y
*/

times(0,X,0).

times(s(X),Y,Z) :-
  times(X,Y,XY),
  plus(XY,Y,Z).
```

Módulo: não recursivo

Código

```
/*
mod(X,Y,Z)
  Z e o resto da divisao inteira de X por Y
*/

mod(X,Y,Z) :-
  times(Y,Q,QY),
  plus(QY,Z,X),
  less(Z,Y).
```

Módulo: recursivo

Código

```
/*
mod(X,Y,Z)
  Z e o resto da divisao inteira de X por Y
*/

mod(X,Y,X) :-
  less(X,Y).

mod(X,Y,Z) :-
  plus(X1,Y,X),
  mod(X1,Y,Z).
```

Considerações gerais sobre listas

- As listas são estruturas binárias enquanto que o o functor sucessor (`s(s(...(0)))`) é unário.
- O primeiro argumento de uma lista tem um *elemento* e o segundo argumento contém recursivamente o resto da lista.
- A lista vazia permite terminar a recursão. É denotada por `[]` e referida como *nil*.
- Historicamente o functor de aridade 2 é ".". Devido à sobreutilização de tal functor, o termo `.(X,Y)` é denotado por `[X|Y]` (`X` é cabeça da lista e `Y` é o corpo). O termo `.(a,[])` é denotado por `[a|[]]` ou de um modo mais simplificado por `[a]`.

Definição de lista

Código

```
/*
lista(Xs)
  Xs e uma lista
*/

lista([]).

lista([X|Xs]) :-
  lista(Xs).
```

Membro de uma lista

Código

```
/*
membro(Elemento, Lista)
   Elemento e um elemento da lista Lista.
*/

membro(X,[X|Xs]).

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

Exemplo

```
membro(X,[a,b,c])?
membro(b,X)?
```

Prefixo de uma lista

Código

```
/*
prefixo(Prefixo,Lista)
  Prefixo e prefixo da lista Lista
*/

prefixo([],Xs).

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

Sufixo de uma lista

Código

```
/*
sufixo(Sufixo,Lista)
  Sufixo e sufixo da lista Lista
*/

sufixo(Xs,Xs).

sufixo(Xs,[Y|Ys]) :-
  sufixo(Xs,Ys).
```

Sublista de uma lista

Código

```
/*
sublista(Sub,Lista)
  Sub e sublista da lista Lista
*/

sublista(Xs,Ys) :-
  prefixo(Ps,Ys),
  sufixo(Xs,Ps).

sublista(Xs,Ys) :-
  sufixo(Ss,Ys),
  prefixo(Xs,Ss).
```

Sublista de uma lista (recursiva)

Código

```
/*
sublista(Sub,Lista)
  Sub e sublista da lista Lista
*/

sublista(Xs,Ys) :-
  prefixo(Xs,Ys).

sublista(Xs,[Y|Ys]) :-
  sublista(Xs,Ys).
```

Concatenação de listas

Código

```
/* concatena(Xs,Ys,XsYs)
   XsYs e o resultado de concatenar
   as listas Xs e Ys */

concatena([],Ys,Ys).

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

Exemplo

```
concatena([1,2],[3,4], X)?
concatena(X,[3,4],[1,2,3,4])?
```

Utilizando o `concatena/3` para definir predicados

Exemplo

```
prefixo(Xs,Ys) :- concatena(Xs,Zs,Ys).
sufixo(Xs,Ys) :- concatena(Zs,Xs,Ys).
membro(X,Ys) :- concatena(As,[X|Xs],Ys).
ultimo(X,Xs) :- concatena(Ys,[X],Xs).
adjacente(X,Y,Zs) :- concatena(As,[X,Y|Ys],Zs).
```

Inverte: *naive*

Código

```
/*
inverte(Lista,Atsil)
 Atsil e o resultado de inverter a Lista
*/
:- ensure_loaded(concatena).

inverte([],[]).

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

Inverte: *non naive*

Código

```
/*
inverte(Lista,Atsil)
 Atsil e o resultado de inverter a Lista
*/
inverte(Xs,Ys) :-
  inverte(Xs,[],Ys).

inverte([],Acc,Acc).

inverte([X|Xs],Acc,Zs) :-
   inverte(Xs,[X|Acc],Zs).
```

Tamanho de uma lista

Código

```
/*
tamanho(Lista,N)
 A lista Lista tem N elementos
 em que N e um numero natural
*/

tamanho([],0).

tamanho([X|Xs],s(N)) :-
   tamanho(Xs,N).
```

Remoção de uma ocorrência de um elemento na lista

Código

```
/* selecciona(X,Lista,ListaSem1X)
 ListSem1X resulta de "remover" uma
 ocorrencia de X na lista Lista*/

selecciona(X,[X|Xs],Xs).

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

Como é definida a remoção de todas as ocorrências?

Código

```
/* ordena(Xs,Ys)
a lista Ys e uma permutacao
ordenada de Xs */

ordena(Xs,Ys) :-
        permutacao(Xs,Ys),
        ordenado(Ys).

permutacao([],[]).

permutacao(Xs,[Z|Zs]) :-
        selecciona(Z,Xs,Ys),
        permutacao(Ys,Zs).
```

Ordenação (*naive*)

Código

```
ordenado([]).

ordenado([_]).

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

Insertion Sort

Código

```
/* ins_sort(Xs,Ys)
a lista Ys e uma permutacao
ordenada de Xs */

insertion_sort([],[]).

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

Código

```
insert(X,[],[X]).

insert(X,[Y|Ys],[Y|Zs]) :-
        less(Y, X),
        insert(X,Ys,Zs).

insert(X,[Y|Ys],[X,Y|Ys]) :-
      less_or_equal(X, Y).
```

Definição de árvore binária

Código

```
/* binary_tree(Tree)
Tree is a binary tree */

binary_tree(void).

binary_tree(tree(_Tree, Left, Right)) :-
        binary_tree(Left),
        binary_tree(Right).
```

Membro de uma árvore binária

Código

```
/* tree_member(Element,Tree)
   Element is an element of the
   binary tree Tree */

tree_member(X,tree(X,_,_)).

tree_member(X,tree(_,Left,_)) :-
        tree_member(X,Left).

tree_member(X,tree(_,_,Right)) :-
        tree_member(X,Right).
```

Substituição de todas as ocorrências de um elemento

Código

```
/* substitute(X,Y,TreeX,TreeY)
 TreeY results from replacing all
 occurrences of X in TreeX by Y*/

substitute(_,_,void,void).

substitute(X,Y,tree(N,L,R),tree(N1,L1,R1)) :-
        replace(X,Y,N,N1),
        substitute(X,Y,L,L1),
        substitute(X,Y,R,R1).

replace(X,Y,X,Y).

replace(X,_Y,Z,Z) :-
        X \= Z.
```

Percorrer uma árvore binária (pré-ordem)

Código

```
/* preorder(Tree,Pre)
 Pre is a preorder traversal of Tree */

preorder(void,[]).
preorder(tree(X,L,R),Xs) :-
        preorder(L,Ls),
        preorder(R,Rs),
        append([X|Ls],Rs,Xs).
```