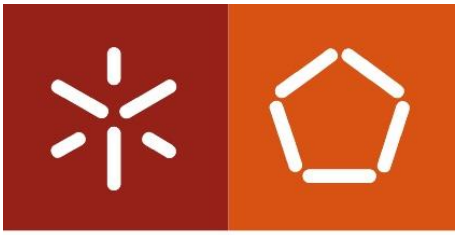

Universidade do Minho

Departamento de Informática

Mestrado em Engenharia Informática

Métodos Formais de Programação

Verificação Formal
Teste Why3

Ariana Lousada (PG47034)

10 de abril de 2022

1 Exercício 2

A solução proposta no momento de avaliação presencial foi a seguinte:

```
let function tree_to_list (t : tree int) : list int
    requires { sortedBT t }
    ensures { sorted result }
    ensures { size t == length result }
    ensures { forall k : int. num_occ t k = number_off k result }
    ensures { memt t k -> memberof k result }
    invariant { length result <= size t }
```

Com exceção de alguns erros de sintaxe, como a utilização de '==' em vez de '=', a falta de um `forall` na última pós-condição e a invocação incorreta das funções pertencentes à biblioteca `List`, o raciocínio parece estar correto:

Uma vez que o enunciado refere explicitamente a transformação de árvores **ordenadas**, é necessário inserir uma pré-condição como

```
requires { sortedBT t }
```

utilizando uma nas funções já abordadas no exercício de árvores apresentado nas aulas da unidade curricular. Neste caso utiliza-se o predicado `sortedBT` que testa se uma dada árvore `t` está ordenada.

Para além disto, o output desta função também tem de ser ordenado. Para isto, invoca-se a função `sorted` da biblioteca `List`. Assim, tem-se a primeira pós-condição:

```
ensures { sorted result }
```

De seguida, definiu-se uma pós-condição que descreve que o tamanho da árvore de *input* tem de ser equivalente ao tamanho da lista resultante. Para isto, invocou-se a função `size` definida para o cálculo do tamanho de árvores no exercício abordado nas aulas da UC e a função `length` da biblioteca `List`:

```
ensures { size t = length result }
```

Para a terceira pós-condição, definiu-se que o número de ocorrências de qualquer elemento pertencente à árvore de *input* tem de ser equivalente ao número de ocorrências do mesmo elemento na lista resultante. Para isto, invocou-se a função `num_occ` já definida para o exercício das árvores, assim como a função `num_occ` da biblioteca `List`:

```
ensures { forall k : int. num_occ1 t k = num_occ k result  }
```

Para a última pós-condição, definiu-se que para qualquer membro da árvore de input, esse mesmo tem de pertencer à lista resultante. Para isto, invocou-se a função `memt` já definida para o exercício das árvores, assim como a função `mem` da biblioteca `List` [1]:

```
ensures { forall k : int. memt t k = mem k result  }
```

Para terminar definiu-se um invariante, após a elaboração de um "esboço"do ciclo da função. Este ciclo vai consistir na constante adição de membros à lista resultado; com isto em mente, o tamanho da lista resultado vai aumentando ao longo do ciclo, até que chega ao próprio tamanho da árvore de input:

```
invariant { 0 <= length result <= size t }
```

Com isto, é proposto como solução desta questão o seguinte:

```
let function tree_to_list (t : tree int) : list int
    requires { sortedBT t }
    ensures { sorted result }
    ensures { size t = length result }
    ensures { forall k : int. num_occ1 t k = num_occ k result }
    ensures { forall k : int. memt t k -> mem k result }
    invariant { 0 <= length result <= size t }
```

[1] Para ver definições das funções realtivas ao exercício das árvores, consultar o anexo A.

2 Exercício 3

A solução proposta no momento de avaliação presencial foi a seguinte:

```
let rec function numof (u : array int) (x : int) (k : int) (l : int)
    requires { l > k }
    ensures { 0 <= result <= l-k }
    variant { l - result }

    for i = k to l-1 do
        invariant { 0 <= result <= l-i }
        if u[i] = x then 1 + numof u x i l
        else numof u x i l
    done;
end;
```

Após inserção e análise desta solução na ferramenta Why3, chegou-se à conclusão que esta solução está completamente errada em termos de sintaxe/utilização de ciclos/utilização de recursividade.

Uma vez que se pretende que esta função seja recursiva, não são necessários ciclos imperativos.

Com isto, a nova proposta de solução a esta questão é a seguinte:

```
let rec function numof (u : array int) (x : int) (k : int) (l : int)
    =
    requires { l >= k }
    ensures { 0 <= result <= l-k }
    variant { l - k }

    if u[k] = x then 1 + numof u x (k+1) l
    else numof u x (k+1) l

end
```

A pré-condição define que o índice l(superior) tem de ser maior ou igual que o índice k(inferior), uma vez que a função não funciona de outra forma.

Uma vez que a variável k vai sendo incrementada, pode-se definir o variante da função como

```
variant { l - k }
```

Relativamente à definição da função em si, esta apenas testa elemento a elemento a sua igualdade com `x`, chamando recursivamente a função para o elemento seguinte incrementando também o resultado caso essa igualdade se verifique.

3 Exercício 4

A solução proposta no momento de avaliação presencial foi a seguinte:

```
let most_frequent (a: array int) : int
    ensures { memberof result a }
    ensures { 1 <= numof a result 0 (length a) <= length a }
    =
    let ref r = a[0] in
    let ref c = 1 in
    let ref m = 1 in
    for i = 1 to length a - 1 do
      invariant { 1 <= c <= length a - 2 }
      if a[i] = a[i - 1] then begin
        incr c;
        if c > m then begin m <- c; r <- a[i] end
      end else
        c <- 1
    done;
    r
end
```

É pretendido que a solução para esta questão funcione para qualquer array de inteiros, não só para arrays ordenados.

No momento da avaliação presencial o objetivo era invocar uma função que testasse se um dado inteiro pertencia a um array. Contudo, na biblioteca `Array` não existe tal função, daí ser necessário defini-la:

```
let function memberof (x : int) (a : array int) : int
    =
        let ref c = 0 in
        for i = 0 to length a - 1 do
            if a[i] = x then c <- 1 else c <- 0
        done;
        c
end
```

Esta função retorna 1 caso o inteiro x pertença ao array a e 0 caso contrário.

Assim, pode-se definir a primeira pós-condição, que define que o resultado(o membro mais frequente do array) seja membro do array de *input*.

```
ensures { memberof result a = 1 }
```

Para a segunda condição definiu-se que o resultado do número de ocorrências do elemento mais frequente no array têm de ser no mínimo 1 e no máximo o tamanho total do array. Para isto invocou-se a função `numof` definida no exercício anterior:

```
ensures { 1 <= numof a result 0 (length a) <= length a }
```

Por fim, definiu-se um invariante para o ciclo. Uma vez que a variável `c` é responsável pela contagem das ocorrências de cada elemento, esta não pode exceder o tamanho do array:

```
invariant { 1 <= c <= length a }
```

Com isto, é proposto como solução desta questão o seguinte:

```
1  let function memberof (x : int) (a : array int) : int
2      =
3      let ref c = 0 in
4      for i = 0 to length a - 1 do
5          if a[i] = x then c <- 1 else c <- 0
6      done;
7      c
8  end
9
10 let most_frequent (a: array int) : int
11     ensures { memberof result a = 1 }
12     ensures { 1 <= numof a result 0 (length a) <= length a }
13     =
14     let ref r = a[0] in
15     let ref c = 1 in
16     let ref m = 1 in
17     for i = 1 to length a - 1 do
18       invariant { 1 <= c <= length a }
19       if a[i] = a[i - 1] then begin
20         incr c;
21         if c > m then begin m <- c; r <- a[i] end
22       end else
23         c <- 1
24     done;
25     r
26   end
```

A Ficheiro para a ferramenta Why3

```
(* Exercicio 2 *)
theory Teste

  use int.Int
  use list.List
  use list.Length
  use list.Permut
  use list.Append
  use list.SortedInt
  use list.Sorted
  use list.NumOcc
  use list.Mem
  use ref.Refint
  use array.Array
  use array.NumOfEq
  use array.IntArraySorted


  type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)

  let rec function size (t : tree int) : int
    ensures { result >= 0 }
    =
    match t with
      | Empty -> 0
      | Node t1 x t2 -> 1 + size t1 + size t2
  end

  let rec predicate memt (t : tree int) (v : int)
    =
    match t with
      | Empty -> false
      | Node t1 k t2 -> if v = k then true
                        else if v > k then memt t2 v else memt t1 v

   end

   let rec function num_occ1 (t : tree int) (v : int) : int
      ensures { result >= 0 }
      =
      match t with
          | Empty -> 0
          | Node t1 k t2 -> if k=v then 1 + num_occ1 t1 v + num_occ1 t2 v
                            else num_occ1 t1 v + num_occ1 t2 v
   end

   (*
   let rec predicate sortedBT (t : tree int)
      =
      match t with
        | Empty -> true
        | Node (Empty) k (Empty) -> true
        | Node (Node t1 k t2) x (Empty) -> k <= x && sortedBT (Node t1 k t2)
        | Node (Empty) x (Node t1 k t2) -> x <= k && sortedBT (Node t1 k t2)
        | Node (Node t1 x1 t2) x (Node t3 x2 t4) -> x1 <= x <= x2 &&
                                                    sortedBT (Node t1 x1 t2) &&
                                                       sortedBT (Node t3 x2 t4)
   end



  val function tree_to_list (t : tree int) : list int
    requires { sortedBT t }
    ensures { sorted result }
    ensures { size t = length result }
    ensures { forall k : int. num_occ1 t k = num_occ k result }
    ensures { forall k : int. memt t k -> mem k result }
    (*variant { size t - length result } *) *)

  (*Exercicio 3*)
  let rec function numof (u : array int) (x : int) (k : int) (l : int)
```

```
72      =
73      requires { l >= k }
74      ensures { 0 <= result <= l-k }
75      variant { l - k }
76
77      if u[k] = x then 1 + numof u x (k+1) l
78      else numof u x (k+1) l
79
80  end
81
82
83
84  (*Exercicio 4*)
85  (*
86  let function memberof (x : int) (a : array int) : int
87    =
88    let ref c = 0 in
89    for i = 0 to length a - 1 do
90        if a[i] = x then c <- 1 else c <- 0
91    done;
92    c
93  end *)
94
95  (*
96  let function most_frequent (a: array int) : int
97    ensures { memberof result a = 1 }
98    ensures { 1 <= numof a result 0 (length a) <= length a }
99    =
100   let ref r = a[0] in
101   let ref c = 1 in
102   let ref m = 1 in
103   for i = 1 to length a - 1 do
104     invariant { 1 <= c <= length a }
105     if a[i] = a[i - 1] then begin
106       incr c;
107       if c > m then begin m <- c; r <- a[i] end
108     end else
109       c <- 1
110   done;
111   r
112 end  *)
```