# Teste Why3 - Parte 2

Henrique Gabriel dos Santos Neto - PG47238

## Exercício 1

O exercício contem dois erros que afectam somente a sintaxe do que está escrito, que são resultantes de eu ter escrito a implementação com a nomenclatura do pseudo-código. Para ambas as variáveis usei os nomes do pseudo-código ($a$ e $b$) em vez do nomes dos argumentos da função ($u$ e $v$). Estou a fazer a chamada recursiva com o nome da função do pseudo-código *gcd* em vez do nome com que a função está a ser declarada *euclid*. Esta é também por esta razão que na pós-condição a função *gcd* importada do modulo *Gcd* é especificada como "*Gcd.gcd*". Com a exceção destes erros, a pré-condição, a pós-condição e o variante estão correctos e para tal o *why3* conseguiu prová-los.

O modulo do exercício para o *why3* já com as correcções da sintaxe é:

```
module EuclideanAlgorithm

    use import int.Int
    use import int.EuclideanDivision
    use import ref.Refint
    use import number.Gcd

    let rec euclid (u v : int) : int
        requires {u > 0 /\ v > 0 }
        ensures  {result = gcd u v}
        variant  {u + v}
    = if u = v then u else (
         if u > v then euclid (u-v) v else euclid u (v-u)
         )
end
```

## Exercício 2

Como agora posso consultar o conteúdo da *stdlib*, posso confirmar que existe de facto existe a função *num_occ* para as listas no modulo *list.NumOcc*, por isso não era necessário defini-la no teste. As funções e os contratos sobre *BTrees* que foram usadas para as provas estão presentes no anexo A. Na resolução do teste existiam erros ao importar os modules, por exemplo o module para o predicado *sorted* é *SortedInt* e não *Sorted*, a definição de ++ está no modulo *Append* e não é contida no modulo base *List* como supus.

O modulo já com as devidas correcções nos *use* é:

```
module Exercicio2

    use int.Int
    use list.List
    use list.NumOcc
    use list.SortedInt
    use list.Append

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

    (* ... Anexo A ... *)

    let rec function tree_to_list (t:tree int) : list int
        requires {sortedBT t}
        ensures  {sorted result}
        ensures  {forall x : int. num_occ x t = NumOcc.num_occ x result}
        variant  { t }
    = match t with
        | Empty -> Nil
        | Node t1 x t2 -> (tree_to_list t1) ++ (Cons x (tree_to_list t2))
      end

end
```

# Exercício 4

Ao contrário do que eu escrevi no teste não é necessário especificar o variante de ciclo para este caso. Adicionalmente, nos invariantes referi-me aos valores de $i$ como $!i$ , porém isto não é correto, e estes são antes acedidos normalmente com $i$.

Por fim para provar as especificações é necessário especificar um invariante de ciclo muito semelhante à ultima pós-condição que não foi especificado durante o teste. Este indica que para qualquer inteiro $x$, o nº de ocorrências de x no intervalo de 0 a $i$ é menor ou igual ao nº de ocorrências de $r$ no intervalo de 0 a $i$, ou seja `forall x : int. (numof a x 0 i) <= (numof a r 0 i)`

Com isto o *why*3 consegue então provar a especificação, sendo esta descrita por:

```
module Exercicio4

    use int.Int
    use ref.Refint
```

```
5       use array.Array
6       use array.NumOfEq
7       use array.IntArraySorted
8
9       let most_frequent (a : array int) : int
10          requires {length a > 0} (* senão r pode ser invalido no inicio*)
11          requires {sorted a}
12          ensures  { (old a) = a}
13          ensures  { forall x : int.
14                       (numof a x 0 (length a))  <= (numof a result 0 (length a)) }
15      = let ref r = a[0] in
16        let ref c = 1 in
17        let ref m = 1 in
18        for i = 1 to length a - 1 do
19          invariant {m = numof a r 0 i}
20          invariant {c = numof a (a[i-1]) 0 i}
21          invariant {forall x : int. (numof a x 0 i)  <= (numof a r 0 i)}
22          if a[i] = a[i-1] then begin
23              incr c;
24              if c > m then begin m <- c ; r <- a[i] end
25          end else
26              c <- 1
27          done;
28          r
29
30  end
```

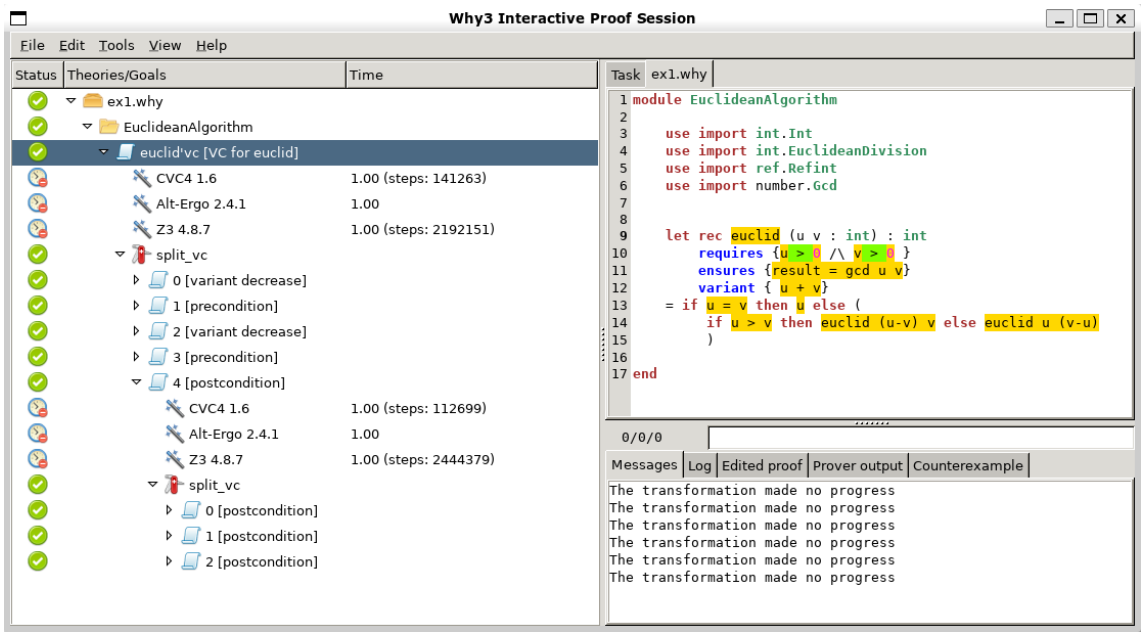
# A Funções e predicados sobre *BTrees*

Funções e contratos provenientes das aulas usados no exercício 2.

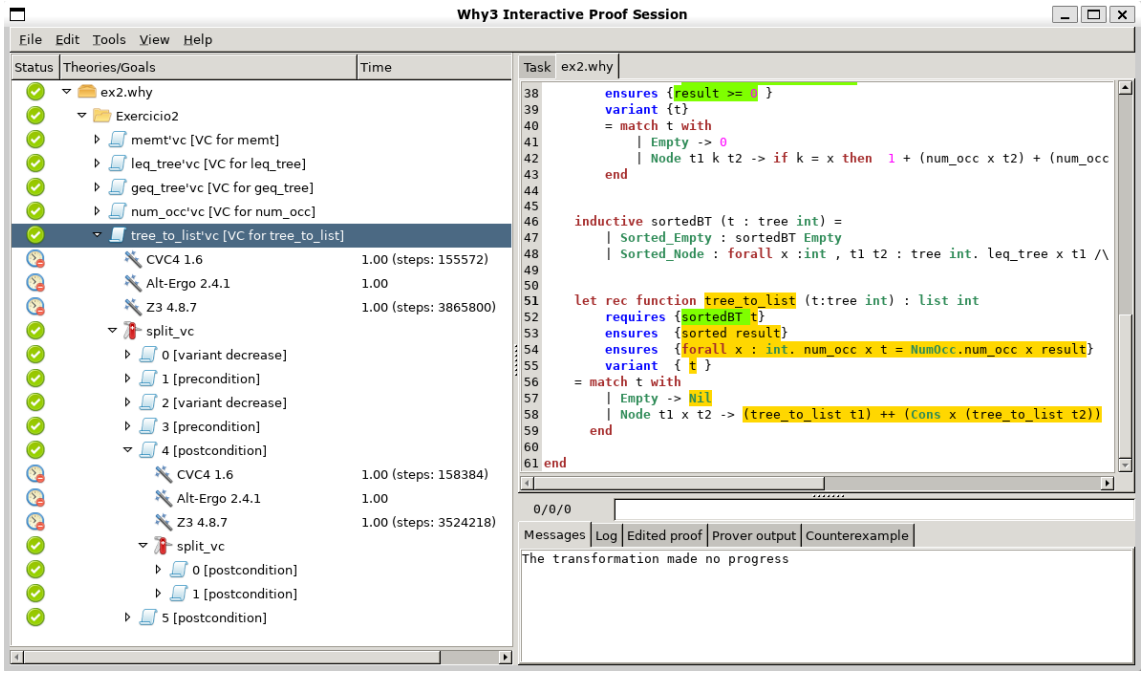
```
1   let rec predicate memt  (t : tree int) (x : int)
2       variant { t }
3       = match t with
4           | Empty -> false
5           | Node t1 k t2 -> if x = k then true else memt t2 x || memt t1 x
6       end
7
8   let rec predicate leq_tree (x : int) (t : tree int)
9       ensures { result <-> forall k : int. not (memt t k) \/ k <= x }
10      variant { t }
11      = match t with
12          | Empty -> true
13          | Node t1 k t2 -> k <= x && leq_tree x t1 && leq_tree x t2
14      end
15
16  let rec predicate geq_tree (x : int) (t : tree int)
17      ensures { result <-> forall k : int. not (memt t k) \/ k >= x }
18      variant { t }
19      = match t with
20          | Empty -> true
21          | Node t1 k t2 -> k >= x && geq_tree x t1 && geq_tree x t2
22      end
23
24  let rec function num_occ (x : int) (t : tree int) : int
25      ensures { result > 0 <-> memt t x }
26      ensures { result >= 0 }
27      variant { t }
28      = match t with
29          | Empty -> 0
30          | Node t1 k t2 -> if k = x then  1 + (num_occ x t2) + (num_occ x t1)
31                            else (num_occ x t2) + (num_occ x t1)
32      end
33
34  inductive sortedBT (t : tree int) =
35      | Sorted_Empty : sortedBT Empty
36      | Sorted_Node : forall x :int , t1 t2 : tree int. leq_tree x t1 /\
37                                      geq_tree x t2 /\ sortedBT t1 /\
38                                      sortedBT t2 -> sortedBT (Node t1 x t2)
```

# B *Print Screens* das sessões de provas

## B.1 Exercício 1




## B.2 Exercício 2



## B.3 Exercício 4