



Universidade do Minho

Escola de Engenharia

Mestrado em Engenharia Informática

Unidade Curricular de Verificação Formal

Ano Letivo 2021/2022

Implementação em Why3 dos Exercícios do Teste

PG47381 José Pedro Ribeiro Peixoto

9 de abril de 2022

VF

Exercício 1

Código escrito na folha de resposta

```
1 let rec function euclid (u v : int) : int
2   requires {u > 0 /\ v > 0}
3   ensures { result = gcd u v }
4   ensures { result ≤ u }
5   ensures { result ≤ v }
6   ensures { result > 0 }
7   = if u <> v then u
8     else
9       if u > v then euclid (u - v) v
10      else euclid u (v - u)
```

Ajustes ao código

Primeiro tive de acrescentar código relativo à importação da biblioteca que exportava a função `gcd` (algo que já estava incluído no enunciado da questão), bem com a teoria sobre inteiros.

```
use int.Int
use number.Gcd
```

Além disso, reparei logo uma confusão com a sintaxe do `why3`... Aquando à resolução do teste, escrevi `<>` pensando que se tratava do sinal de igualdade. Sendo assim, troquei pelo verdadeiro sinal de igualdade `=`.

Antes

```
= if u <> v then u
```

Depois

```
= if u = v then u
```

Para terminar, reparei num problema crucial e que faz toda a diferença para conseguir provar a correção da função: a terminação do ciclo através do variante. Se repararmos no código, por vezes é o `u` que decresce, noutras vezes é o `v`... mas nós sabemos intuitivamente que isto termina... mas como?

Entretanto apercebi-me onde se funda esta intuição: estes números vão-se aproximando de 0, ou seja **a soma do seu valor absoluto vai diminuindo**, neste caso, como são sempre positivos, basta que o variante seja `u + v`.

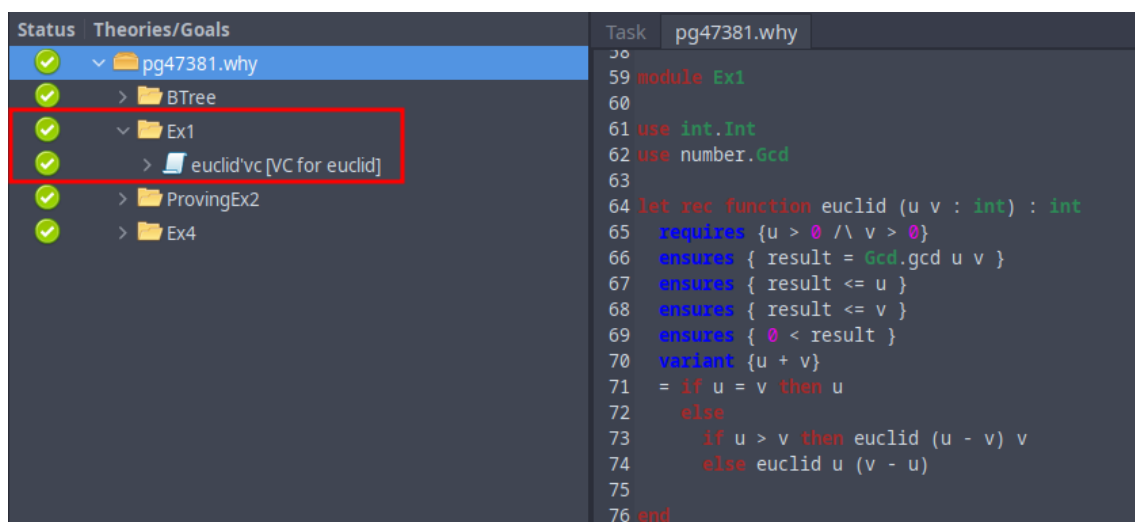
Sendo assim, chega-se ao seguinte código:

```

1 use int.Int
2 use number.Gcd
3
4 let rec function euclid (u v : int) : int
5   requires {u > 0 /\ v > 0}
6   ensures { result = Gcd.gcd u v }
7   ensures { result ≤ u }
8   ensures { result ≤ v }
9   ensures { 0 < result }
10  variant {u + v}
11  = if u = v then u
12    else
13      if u > v then euclid (u - v) v
14      else euclid u (v - u)

```

Que conseguiu ser provado pelo why3:



The screenshot shows the Why3 IDE interface. On the left, the 'Theories/Goals' pane lists the project structure: 'pg47381.why' (expanded), 'BTree', 'Ex1' (expanded), 'euclid\'vc [VC for euclid]' (highlighted with a red box), 'ProvingEx2', and 'Ex4'. All items have a green checkmark indicating they are proven. On the right, the 'Task' pane shows the source code for 'pg47381.why', which includes the 'euclid' function definition and its associated preconditions and postconditions.

Exercício 2

Código escrito na folha de resposta

```

1 let function tree_to_list (t : tree int) : list int
2   requires {sortedBT t}
3   ensures {sorted result}
4   ensures {forall x : int. num_occBT x t = num_occ x result}
5   ensures {sizeBT t = size result}

```

Ajustes ao código

Obviamente que o `why3` não pode provar nada em relação a esta função, uma vez que não está declarada.

Então o que fiz foi dividir o problema em duas partes:

1. Construir um contrato à parte para esta função
2. Definir uma função que cumprisse o contrato e mostrar a sua correção

Para mais tarde dizer ao `why3` que esta função é um refinement do contrato, separei estas duas partes em dois módulos distintos.

Definição do Contrato

Obviamente que a definição do contrato é essencialmente a cópia do já foi escrito na minha resposta ao teste. Tal como no exercício 1, importei as bibliotecas necessárias, inclusive a minha resolução das `binary trees` da ficha de aulas anteriores, que deixo nos anexos deste relatório.

```
1 use list.List
2 use list.SortedInt
3 use list.NumOcc
4 use list.Length
5 use BTree
6
7 val function tree_to_list (t : tree int) : list int
8   requires {sortedBT t}
9   ensures {sorted result}
10  ensures {forall x : int. num_occBT x t = num_occ x result}
11  ensures {sizeBT t = length result}
```

Definição da função

Primeiro para mostrar que a função cumpria o contrato, voltei a declarar as mesmas pré-condições e pós-condições.

Depois, defini a função, utilizando a biblioteca `list.Append` por causa da concatenação de listas, necessária para o funcionamento da função.

Uma vez que a minha implementação é recursiva (`rec`), então também acrescentei o variante, que neste caso é `t`.

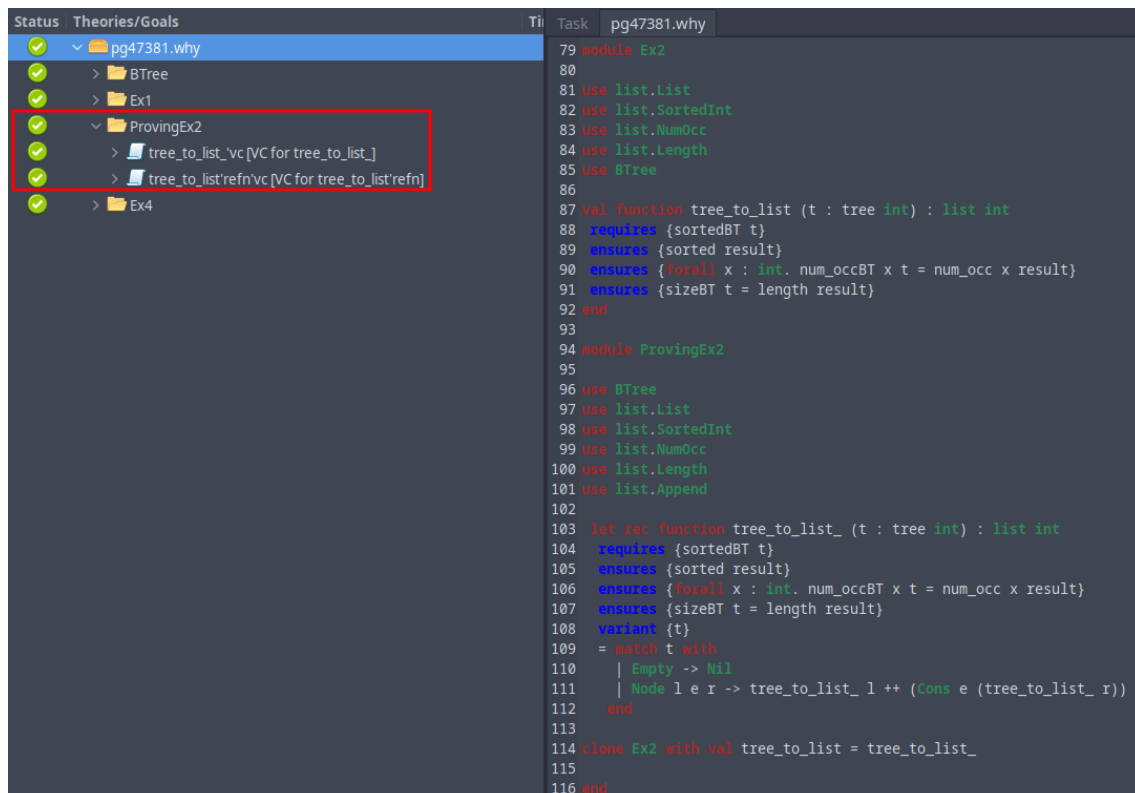
Obtendo:

```

1 use BTree
2 use list.List
3 use list.SortedInt
4 use list.NumOcc
5 use list.Length
6 use list.Append
7
8 let rec function tree_to_list_ (t : tree int) : list int
9   requires {sortedBT t}
10  ensures {sorted result}
11  ensures {forall x : int. num_occBT x t = num_occ x result}
12  ensures {sizeBT t = length result}
13  variant {t}
14  = match t with
15    | Empty -> Nil
16    | Node l e r -> tree_to_list_ l ++ (Cons e (tree_to_list_ r))
17  end
18
19 clone Ex2 with val tree_to_list = tree_to_list_

```

Com isto, provei a sua correção no why3:



Exercício 4

Código escrito na folha de resposta

```
1 let most_frequent (a : array int) : int
2   requires {length a > 0}
3   requires {forall i j : int. 0 ≤ i < j < (length a) → a[i] ≤ a[j] }
4   ensures { forall x : int. (numof a x 0 (length a)) ≤ (numof a result 0 (length
   ↪ a)) }
5   ensures { (numof a result 0 (length a)) > 0}
6   = let ref r = a[0] in
7     let ref c = 1 in
8     let ref m = 1 in
9     for i = 1 to length a - 1 do
10      invariant {(forall j : int. 0 ≤ j < i → (numof a a[j] 0 i) ≤ !m /\ !m =
   ↪ numof a !r 0 i /\ !c ≤ m) /\ !m ≤ length a}
11      if a[i] = a[i-1] then begin
12        incr c;
13        if c > m then begin m ← c ; r ← a[i] end
14      end else
15        c ← 1
16      done;
17      r
```

Ajustes ao código : Parte 1

Tal como no exercício 1, a primeira adição que fiz foi a importação das bibliotecas necessárias. Acrescentado a biblioteca `array.NumOfEq` visto que não resolvi o exercício 3.

O primeiro problema que me apareceu teve, novamente, a ver com sintaxe... devido à biblioteca `ref`, não foi necessária a utilização do `!` sempre que precisava do valor de uma referência e, por isso, omiti-os.

No entanto, o why3 não conseguiu provar a correção da função, mais concretamente, o invariante de ciclo.

Ajustes ao código : Parte 2

Entretanto reparei que não tinha nenhuma condição sobre a variável `c`, então decidi que podia ser um possível caminho a seguir. Mas o que é que não variava em relação ao valor de `c` ?

Sabia o seguinte:

- Se o valor de `i-1` fosse igual a `i`, então `c` é apenas incrementado
- Senão, `c` é 1.

Mas em termos semânticos... o que é que isto significa? Bem, como o **array está ordenado**, `i-1` não ser igual a `i` significa que o elemento em `i-1` nunca mais vai aparecer para a frente do array, uma vez que está ordenado. Sendo assim, quando o `c` passa a 1 significa que está a contar as ocorrências de um novo elemento. Mais concreto que isso, o `c` tem sempre o número de referências do último elemento que explorou, que é `a[i-1]`.

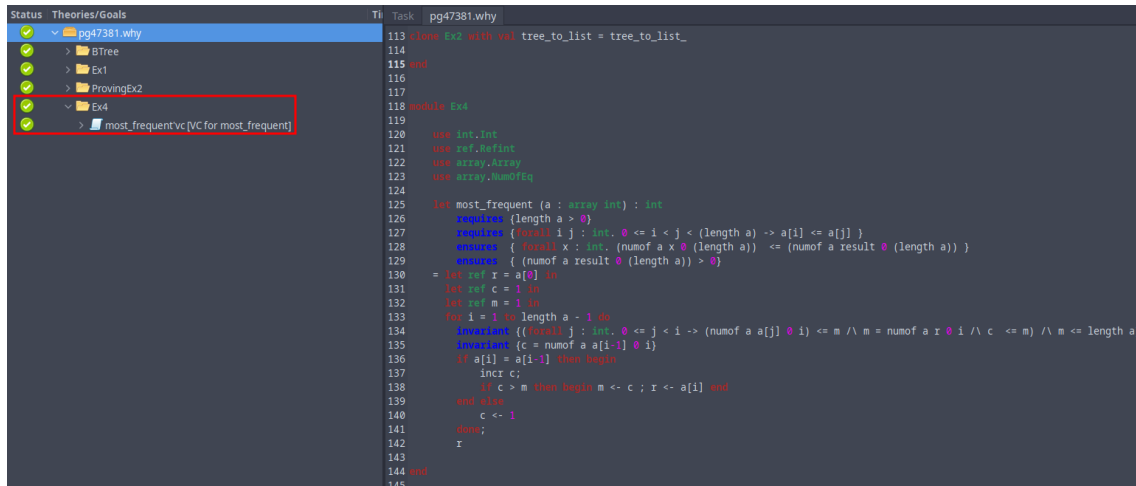
Ou seja, `c` vai sempre ser o número de ocorrências de `a[i-1]` entre 0 e `i`, i.e.:

```
1 invariant {c = numof a a[i-1] 0 i}
```

Resultando o seguinte código:

```
1 use int.Int
2 use ref.Refint
3 use array.Array
4 use array.NumOfEq
5
6 let most_frequent (a : array int) : int
7   requires {length a > 0}
8   requires {forall i j : int. 0 ≤ i < j < (length a) → a[i] ≤ a[j]}
9   ensures { forall x : int. (numof a x 0 (length a)) ≤ (numof a result 0 (length
10     ↪ a)) }
11   ensures { (numof a result 0 (length a)) > 0}
12 = let ref r = a[0] in
13   let ref c = 1 in
14   let ref m = 1 in
15   for i = 1 to length a - 1 do
16     invariant {(forall j : int. 0 ≤ j < i → (numof a a[j] 0 i) ≤ m /\ m = numof a
17       ↪ r 0 i /\ c ≤ m) /\ m ≤ length a}
18     invariant {c = numof a a[i-1] 0 i}
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
```

E agora sim, o why3 já conseguiu provar a correção da função:



Informações Adicionais

Opam, versão: 2.1.2

Why3, versão: 1.4.1

Provers utilizados:

- Alt-ergo, versão: 2.4.1
- CVC4, versão: 1.8
- Z3, versão: 4.8.9 - 64 bits

As bibliotecas referenciadas podem ser encontradas em: <http://why3.lri.fr/stdlib/>

Sistemas operativo utilizado: Arch Linux

Anexos

```

1 module BTree
2
3 use int.Int
4
5 type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)
6
7 let rec predicate memt (t : tree int) (x : int) =
8   variant {t}
9     match t with
10      | Empty → false

```



```

11     | Node l e r → e = x || memt l x || memt r x
12   end
13
14 let rec predicate leq (x : int) (t : tree int) =
15   ensures { result ↔ (forall k : int. memt t k → x ≤ k) }
16   variant {t}
17   match t with
18   | Empty → true
19   | Node l e r → x ≤ e && leq x l && leq x r
20   end
21
22 let rec predicate geq (x : int) (t : tree int) =
23   ensures { result ↔ forall k : int. memt t k → x ≥ k }
24   variant {t}
25   match t with
26   | Empty → true
27   | Node l e r → x ≥ e && geq x l && geq x r
28   end
29
30
31 let rec predicate sortedBT (t : tree int) =
32   variant {t}
33   match t with
34   | Empty → true
35   | Node l e r → sortedBT l && sortedBT r && geq e l && leq e r
36   end
37
38 let rec ghost function sizeBT (t : tree 'a) : int =
39   variant {t}
40   match t with
41   | Empty → 0
42   | Node l _ r → sizeBT l + 1 + sizeBT r
43   end
44
45
46 let rec ghost function num_occBT (x : int) (t : tree int) : int =
47   variant {t}
48   ensures { (memt t x ↔ result > 0) /\ (not (memt t x) ↔ result = 0) }
49   match t with
50   | Empty → 0
51   | Node l e r → (if e = x then 1 else 0) + num_occBT x l + num_occBT x r
52   end
53
54 end

```