Why3 - Ficha 1

Data: 12-10-2022
Tags: #ALGC #uni #SoftwareEngineering #C

Website: https://why3.lri.fr/try/
MicroC.

a)

```
int minInd (int v[], int N) {
    //@ assume N == length(v);
    //@ assume N > 0;
    int i = 1;
    int r = 0;
    while (i<N) {
        //@ invariant 1 <= i <= N;
        //@ invariant (forall k . 0 <= k < i -> v[r] <= v[k]) && (0 <= r < i);
        //@ variant N-i;
        if (v[i] < v[r]) r = i;
        i = i+1;
    }
    //@ assert 0 <= r < N;
    //@ assert forall k . 0 <= k < N -> v[r] <= v[k];
    return r;
}
```

b)

```
int minimo (int v[], int N) {
    //@ assume N == length(v);
    //@ assume N > 0;
    int i = 1;
    int r = v[0];
    while (i!=N) {
        //@ invariant 1 <= i <= N;
        //@ invariant forall k . 0 <= k < i -> r <= v[k];
        //@ invariant exists k . 0 <= k < i -> r == v[k];
        //@ variant N-i;
        if (v[i] < r) r = v[i];
        i=i+1;
    }
    //@ assert forall k . 0 <= k < N -> r <= v[k];
    //@ assert exists k . 0 <= k < N -> r == v[k];
    return r;
}
```

c)

```
int soma (int v[], int N) {
   //@ assume N == length(v);
   //@ assume N >= 0;
   int i = 0;
   int r = 0;
   while (i!=N) {
       //@ invariant 0 <= i <= N;
```

```
        //@ invariant r == sum(v, 0, i);
        //@ variant N-i;
        r = r + v[i];
        i=i+1;
    }
    //@ assert r == sum(v,0,N);
    return r;
}
```

d)

```
int quadrado1 (int x) {
    //@ assume x >= 0;
    int a = x;
    int b = x;
    int r = 0;

    while (a!=0) {
        //@ invariant 0 <= a <= x;
        //@ invariant a*b+r == x*x;
        //@ variant a;
        if (a%2 != 0) r = r + b;
        a = a/2; b = b*2;
    }
    //@ assert r == x*x;
    return r;
}
```

$I \wedge C \implies Variant \geq 0$

Portanto, o invariante $0 \leq a \leq x$ e a condição de ciclo $a \neq 0$ implicam que 'a' decresce mas é sempre maior ou igual a 0.

$$(a \times b + r = x^2 \wedge a \neq 0 \wedge 0 \leq a \leq x)$$
$$\Rightarrow a \geq 0$$
$$(a \times b + r = x^2 \wedge 0 < a \leq x) \quad \text{TRUE}$$
$$= a \geq 0$$

e)

```c
int quadrado2 (int x) {
    //@ assume x >= 0;
    int r = 0;
    int i = 0;
    int p = 1;

    while (i<x) {
        //@ invariant i <= x;
        //@ invariant p == 2*i + 1;
        //@ invariant r == i*i;
        //@ variant x-i;
        i = i+1;
        r = r+p;
        p = p+2;
    }
    //@ assert r == x*x;
    return r;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int quadrado2 (int x) {
    //@ assume x >= 0;
    int r = 0;
    int i = 0;
    int p = 1;

    while (i<x) {
        //@ invariant i <= x;
        //@ invariant p == 2*i + 1;
        //@ invariant r == i*i;
        //@ variant x-i;
        i = i+1;
        r = r+p;
        p = p+2;
    }
    //@ assert r == x*x;
    return r;
}
```

Task list | Task view

- Trywhy3_input
  - VC for quadrado2
    - loop invariant init
    - loop invariant init
    - loop invariant init
    - loop variant decrease
    - loop invariant preservation
    - loop invariant preservation
    - loop invariant preservation
    - assertion

$$(r = i^2 \wedge i < x) \Rightarrow (r = i^2)[p^{p+2}][r^{r+p}]$$
$$(r = i^2 \wedge i < x) \Rightarrow r + p = (i+1)^2 \; [i \backslash i+1]$$
$$\Leftrightarrow (r = i^2 \wedge i < x) \Rightarrow r + p = i^2 \; (2i + 1)$$
$$p = 2i + 1$$

f)

Demorei neste porque me estava a esquecer do caso em que o array está todo ordenado; aí `v[r-1]` não é maior do que `v[r]`. Então adicionei `(r == N)` à pós-condição.

```
int maxPOrd (int v[], int N){
    //@ assume N == length(v);
    //@ assume N > 1;
    int r = 1;
    while (r < N && v[r-1] <= v[r]) {
        //@ invariant 1 <= r <= N;
        //@ invariant forall k . 0 < k < r -> v[k-1] <= v[k];
        //@ variant N-r;
        r = r+1;
    }
    //@ assert (r == N) || (forall k . 0 < k < r -> v[k-1] <= v[k]) && v[r-1] > v[r];
    return r;
}
```

Outra versão:

```
int maxPOrd (int v[], int N){
    //@ assume N == length(v);
    //@ assume N > 1;
    int r = 1;
    while (r < N && v[r-1] <= v[r]) {
        //@ invariant 0 <= r <= N;
        //@ invariant (r > 0 && forall k . 0 < k < r -> v[k-1] <= v[k]) || (r == 0 && N == 0);
        //@ variant N-r;
        r = r+1;
    }
    //@ assert (0 <= r <= N);
    //@ assert (r == 0 -> N == 0);
    //@ assert (r > 0 -> (forall k . 0 < k < r -> v[k-1] <= v[k]));
    //@ assert (r < N -> v[r] < v[r-1]) || r == N;
    return r;
}
```

g)

Basta `x == a[p]`, não é necessário colocar limites ao `p` visto que estes podem ser deduzidos através da prova lógica (a limitação de `p` está presente no primeiro argumento da implicação lógica - no invariante -, ao verificar a Utilidade)

```
int procura1 (int x, int a[], int N) {
    //@ assume N == length(a);
    //@ assume N > 0;
    int p = -1;
    int i = 0;
    while (p == -1 && i < N) {
        //@ invariant 0 <= i <= N;
        //@ invariant (p == -1 && forall k . 0 <= k < i -> a[k] != x) || (0 <= p < i && a[p] == x);
        //@ variant N-i;
        if (a[i] == x) p = i;
        i = i+1;
    }
    //@ assert (p == -1 && forall k . 0 <= k < N -> a[k] != x) || x == a[p];
    return p;
}
```

h)

```
int procura2 (int x, int a[], int N) {
    //@ assume N == length(a);
    //@ assume N > 0;
    //@ assume forall i,j . 0 <= i <= j < N -> a[i] <= a[j];
    int p = -1;
    int i = 0;
    while (p == -1 && i < N && x >= a[i]) {
        //@ invariant 0 <= i <= N;
        //@ invariant (p == i-1 && a[p] == x) || (p == -1 && forall k . 0 <= k < i -> a[k] != x);
        //@ variant N-i;
        if (a[i] == x) p = i;
        i = i+1;
    }
    //@ assert (p == -1 && forall k . 0 <= k < N -> a[k] != x) || (0 <= p < N && x == a[p]);
    return p;
}
```

Tem de ser `p == i-1` visto que depois do `if`, o `i` é incrementado.

i)

https://why3.lri.fr/try/?name=triangular.c&lang=micro-C

Ideias que não funcionaram (mas a acertion fica correta):
`//@ invariant (p == -1 && (forall k . 0 <= k <= i -> a[k] != x) && (forall k . s <= k < N -> a[k] != x));`

`//@ invariant (p == -1 && (forall k . 0 <= k <= i -> a[k] != x) && (forall k . s <= k < N -> a[k] != x)) || (0 <= p < N && x == a[p]);`

`//@ invariant (p == -1 && i <= s && (forall k . 0 <= k <= i -> a[k] != x) && (forall k . s <= k < N -> a[k] != x)) || (p == -1 && i > s && forall k . 0 <= k < N -> a[k] != x) || (0 <= p < N && x == a[p]);`

https://github.com/kit-ty-kate/why3/blob/master/examples/binary_search.mlw

O invariante `0 <= (i+s)/2 < N` afinal é desnecessário, deve ser possível chegar a esses limites através de deduções lógicas com os limites de `i` e de `s`.

Ainda persiste este problema:
loop variant decrease (unknown)

```
// procura binária de um array ordenado
int procura3 (int x, int a[], int N) {
    //@ assume N == length(a);
    //@ assume N > 0;
    //@ assume forall i,j . 0 <= i <= j < N -> a[i] <= a[j];
    int p = -1;
    int i = 0;
    int s = N-1;
    int m;
    while (p == -1 && i <= s) {
        //@ invariant 0 <= i <= N && -1 <= s < N;
        //@ invariant (p == -1 && forall k . 0 <= k < N -> a[k] == x -> i <= k <= s) || (p == (i+s)/2 &&
x == a[p]);
        //@ variant s-i;
        m = (i+s)/2;
        if (a[m] == x) p = m;
        else if (a[m] > x) s = m-1;
        else i = m+1;
    }
    //@ assert (p == -1 && forall k . 0 <= k < N -> a[k] != x) || (0 <= p < N && x == a[p]);
```

```
    return p;
}
```

j)

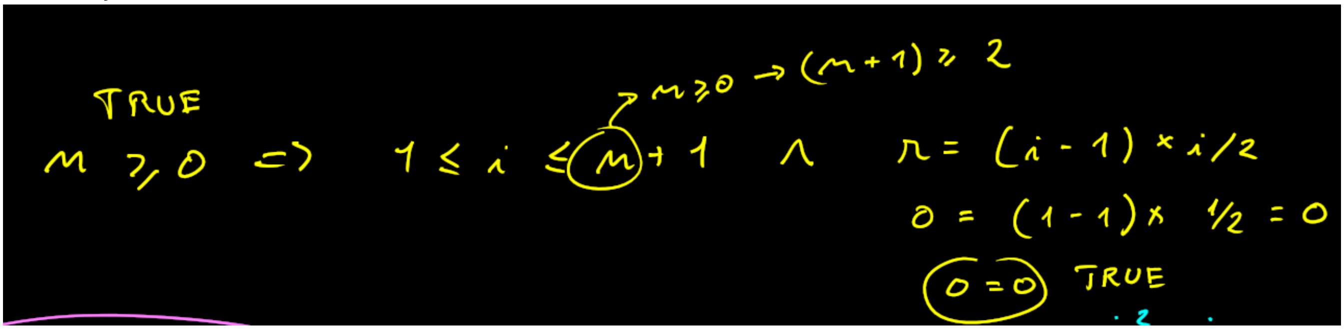
Soma dos primeiros n números inteiros: (versão com parte incorreta)

```
int triangulo1 (int n) {
    //@ assume n >= 0;
    int r=0;
    int i=1;
    while (i!=n+1) {
        //@ invariant 1 <= i <= n+1;
        //@ invariant r == (i-1) * i/2;
        //@ variant n-i;
        r = r+i; i = i+1;
    }
    //@ assert r == n * (n+1) / 2;
    return r;
}
```

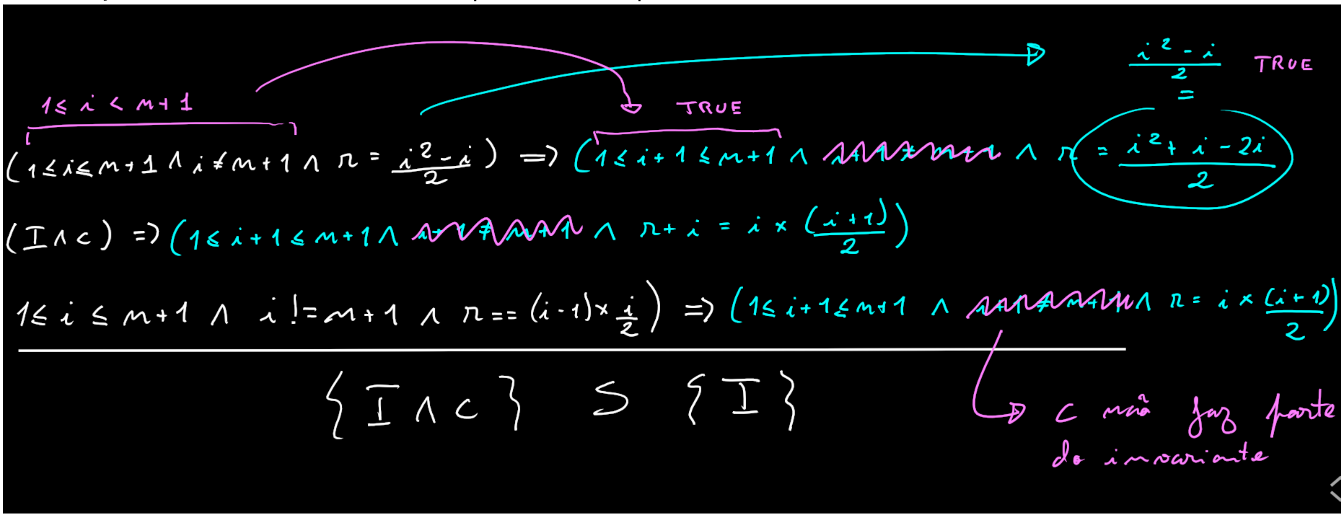
Não sei porquê que não se verifica a preservação, visto que consegui provar formalmente.

loop invariant preservation (unknown)

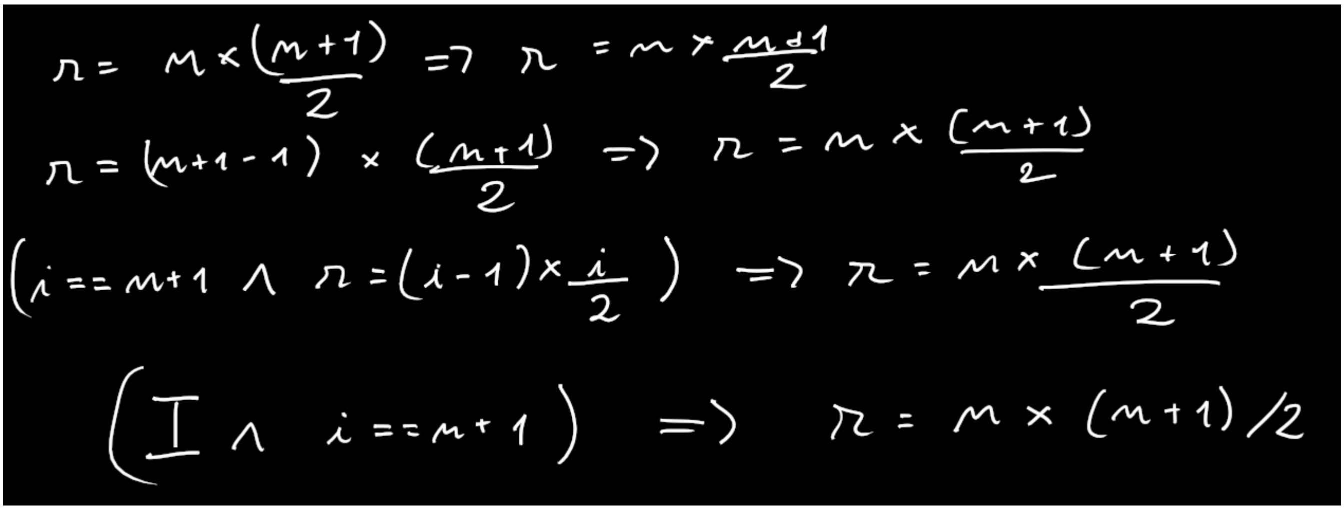
Inicialização:



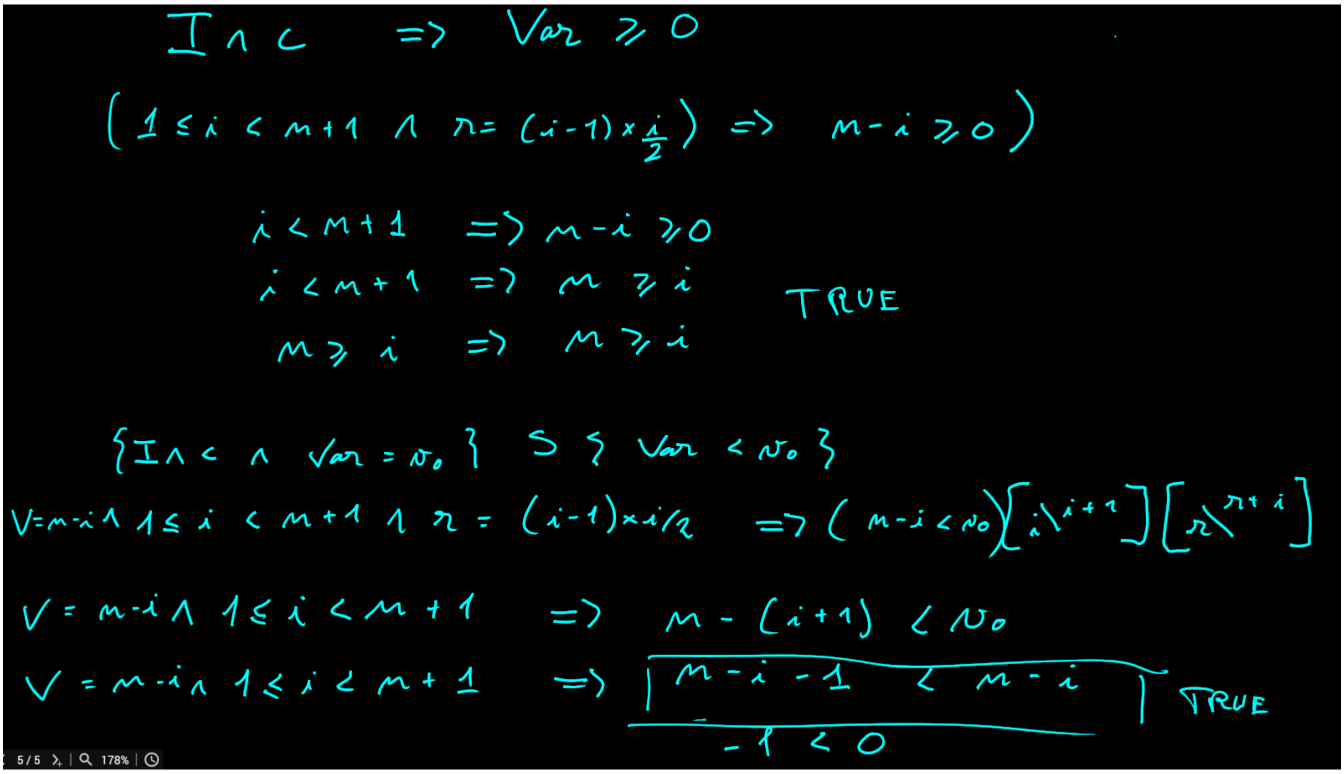
Preservação: (mais abaixo é destacado o que falta nesta prova)



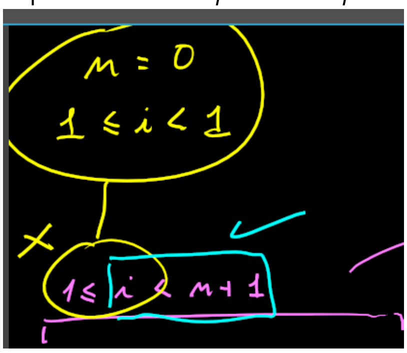
Utilidade:



Variante:



O problema do *loop invariant preservation* em cima era o `1 <= i`.



Versão 100% correta:

```
int triangulo1 (int n) {
    //@ assume n >= 0;
    int r=0;
    int i=1;
    while (i!=n+1) {
        //@ invariant i <= n+1;
        //@ invariant r == (i*i - i)/2;
        //@ variant n-i;
        r = r+i; i = i+1;
```

```
    }
    //@ assert r == n * (n+1) / 2;
    return r;
}
```

Ou, por exemplo, apesar de serem necessários mais passos no why3 para o provar:
(`//@ invariant 0 <= i <= n+1`)

```
int triangulo1 (int n) {
    //@ assume n >= 0;
    int r=0;
    int i=1;
    while (i!=n+1) {
        //@ invariant 0 <= i <= n+1;
        //@ invariant r == (i*i - i)/2;
        //@ variant n-i;
        r = r+i; i = i+1;
    }
    //@ assert r == n * (n+1) / 2;
    return r;
}
```

Outra versão:

$$r = \frac{(i+1)\times(i+1-1)}{2}$$

$$r = \frac{(i-1)i}{2} \qquad r + i = \frac{(i+1)\times i}{2}$$

$$r + i = \frac{(i-1)i}{2} + \frac{2i}{2}$$

$$r + i = \frac{(i-1)\times i + 2i}{2} = \frac{i(i-1+2)}{2} = \frac{i(i+1)}{2}$$

```
int triangulo1 (int n) {
    //@ assume n >= 0;
    int r=0;
    int i=1;
    while (i!=n+1) {
        //@ invariant 1 <= i <= n+1;
        //@ invariant 0 <= r;
```

```
        //@ invariant r + i == (i*(i+1))/2;
        //@ variant n-i;
        r = r+i; i = i+1;
    }
    //@ assert r == n * (n+1) / 2;
    return r;
}
```

k)

$r ==$ somatório com n - somatório com i (100% correto)

```
int triangulo2 (int n){
    //@ assume n >= 0;
    int r=0;
    int i=n;
    while (i>0) {
        //@ invariant i >= 0;
        //@ invariant r == (n*(n+1) - i*(i+1))/2;
        //@ variant i;
        r = r+i; i = i-1;
    }
    //@ assert r == n * (n+1) / 2;
    return r;
}
```

l)

Resto da divisão inteira. (100% correto)

```
//@ lemma aux: forall x,y,r,q . 0 <= r <= x && y > 0 && q >= 0 && x == q*y + r -> x == (q+1)*y + r-y;

int mod (int x, int y) {
    //@ assume x >= 0 && y > 0;
    int r = x;
    while (y <= r) {
        //@ invariant x >= r >= 0;
        //@ invariant exists q . x == q*y + r;
        //@ variant r-y;
        r = r-y;
    }
    //@ assert 0 <= r < y;
    //@ assert exists q . x == q*y + r;
    return r;
}
```

Prova matemática do invariante:

$$x = q \times y + (x - ky)$$

$$0 = y(q - k)$$

$$y = 0 \wedge q = k$$

Portanto, como $y > 0$ (pré-condição), existe sempre um $q = k$.

m)

(Não sei definir funções que possam ser usadas dentro de invariantes no why3 com microC)

```
// Valor de um polinómio num ponto

int pow (int x, int i) {
    for (int j = 0; j < i; i++) {
        x = x * j;
    }
    return x;
}

//@ function int sum1 (int x, int coef[], int N);

int sum1 (int x, int coef[], int N) {
    //@ assume N >= 0;
    int r = 0;
    for (int i = 0; i < N; i++)
        r = r + pow(x,i) * coef[i];
    return r;
}

int valor1 (int x, int coef[], int N) {
    //@ assume N >= 0;
    int r = 0;
    int i = 0;
    while (i < N) {
        //@ invariant sum1(x, coef, i);
        //@ variant N-i;
        r = r + pow(x,i) * coef[i];
        i = i+1;
    }
    //@ assert r == sum1(x, coef, N);
    return r;
}

int main() {
    return 0;
}
```