Trabalho 4

Este trabalho foi realizado por:

- João Pedro Goulart - A82643
- Tiago Rodrigues - A87952

Considere o seguinte programa, em Python anotado, para multiplicação de dois inteiros de precisão limitada a 16 bits.

```
   assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
0: while y > 0:
1:     if y & 1 == 1:
           y , r  = y-1 , r+x
2:     x , y = x<<1  ,  y>>1
3: assert r == m * n
```

1. Prove por indução a terminação deste programa

```
!pip install z3-solver
```

```
Requirement already satisfied: z3-solver in /usr/local/lib/python3.7/dist-packages (4.8.14.0)
```

```
from z3 import *
import math
```

```
def provel(f):
  s = Solver()
  s.add(Not(f))
```

```
    r = s.check()
    if r == unsat:
      print("Provado")
    else:
      print("Não foi possível provar")
      m = s.model()
      for v in m:
        print(v,'=',m[v])
```

```
m, n, r, x, y = Ints('m n r x y')

provel(Implies(And(y<=0, 0<=y, y<=n, m*n == x*y+r), r == m*n))
```

```
Provado
```

```
Inicialização do ciclo

assume m >= 0 and n >= 0 and r == 0 and x == m and y == n; skip; 0<=n<=y and m*n==x*y+r
m >= 0 and n >= 0 -> 0<=n<=y and  m * n == x * y + r [y/n][x/m][r/0]
m >= 0 and n >= 0 -> 0<=n<=n and  m * n == m * n + 0

pos = r == m * n
ciclo = (assume m >= 0 and n >= 0 and r == 0 and x == m and y == n; skip; assert 0<=n<=y
and  m * n = x * y + r) and (assume y>0 and 0<=n<=y and  m * n == x * y + r;
if y & 1 == 1 then y, r = y-1, r+x ; x , y = x<<1  ,  y>>1;  0<=n<=y
and  m * n = x * y + r) and (assume y<=0 and 0<=n<=y and
m * n == x * y + r; skip; assert r == m * n)

[while y>0 : if y & 1 == 1 then y , r  = y-1 , r+x ; assert pos; x , y = x<<1 , y>>1;
assert pos;]

[ciclo; (assume  y & 1 == 1;  y , r  = y-1 , r+x; || assume ~y | 1 != 1;); assert pos;
```

```
x , y = x<<1   ,  y>>1 ; assert pos]

[ciclo; (assume  y & 1 == 1;  y , r  = y-1 , r+x; x , y = x<<1,  y>>1 ; assert pos; ||
assume ~y | 1 != 1; assert pos); x , y = x<<1   ,  y>>1 ; assert pos]

[ciclo; [y & 1 == 1 -> [ y , r  = y-1 , r+x; assert pos;]] or [~y | 1 != 1; assert pos];
x , y = x<<1   ,  y>>1 ; assert pos;]

[ciclo; [y & 1 == 1 -> [ y , r  = y-1 , r+x; assert pos;]] or [~y | 1 != 1; assert pos];
x , y = x<<1   ,  y>>1 ; assert pos;]

[ciclo; [y & 1 == 1 -> [ y , r  = y-1 , r+x; assert pos;]] or [~y | 1 != 1 assert pos];
x , y = x<<1   ,  y>>1 ; assert pos;]

[ciclo; y & 1 == 1 -> [[ y , r  = y-1 , r+x; assert pos;]] or [~y | 1 != 1; assert pos];
x , y = x<<1   ,  y>>1 ; assert pos;]

[ciclo; y & 1 == 1 -> [[y=y-1; r=r+x; assert pos;]] or [~y | 1 != 1; assert pos];
x , y = x<<1   ,  y>>1 ; assert pos]

[ciclo; y & 1 == 1 -> pos[y/(y>>1)][r/r+(x<<1)] or [~y | 1 != 1-> pos];
pos[x/x<<1][y/y<<2];]
```

```
m, n, r, x, y = BitVecs("m n r x y", 16)

pre = And(m >= 0, n >= 0, r == 0, x == m, y == n)
pos = (r == m * n)
inv = And(0<=y,y<=n,m*n==x*y+r)

ifT=Implies(y & 1 == 1, substitute(substitute(substitute(inv, (x, x<<1)), (y, (y-1)>>1)), (r, r+x)))
ifF=Implies(Not(y & 1 == 1),substitute(substitute(inv, (x, x<<1)), (y, y>>1)))
```

```
ciclo = ForAll([x, y, r], Implies(And(y>0, inv), Or(ifT, ifF)))
final = Implies(And(Not(y>0), inv), pos)

prove(Implies(pre, And(inv, ciclo, final)))
```

```
proved
```

Para efetuar esta prova podemos construir um FOTS que modela o programa

```
def declare(i):
    state = {}
    state["x"] = BitVec("x[i]", 16)
    state["y"] = BitVec("y[i]", 16)
    state["r"] = BitVec("r[i]", 16)
    state["m"] = BitVec("m[i]", 16)
    state["n"] = BitVec("n[i]", 16)
    state["pc"] = BitVec("pc[i]",16)

    return state
```

O estado inicial do FOTS é determinado pelo predicado init. olhando para a pré-condição deste programa, o predicado init corresponde a:

$$pc == 0 \land m \geq 0 \land n \geq 0 \land r == 0 \land x == m \land y == n$$

```
def init(state):
    return And(state["pc"]==0, state["m"]>=0, state["n"]>=0, state["r"]==0, state["x"]==state["m"], state["y"]==state["n"]
```

Prosseguindo, determinaremos a função de transição.

```
def trans(curr, prox):
    pc0_1 = And(curr["pc"]==0, curr["y"]>0, prox["x"]==curr["x"], prox["y"]==curr["y"],
                prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"],
                prox["pc"]==1)
```

```
    pc0_2 = And(curr["pc"]==0, Not(curr["y"]>0), prox["x"]==curr["x"], prox["y"]==curr["y"],
                prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"],
                prox["pc"]==3)


    pc1_1 = And(curr["pc"]==1, curr["y"]&1==1, prox["x"]==curr["x"], prox["y"]==curr["y"]-1,
                prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"]+curr["x"],
                prox["pc"]==2)
    pc1_2 = And(curr["pc"]==1, Not(curr["y"]&1==1), prox["x"]==curr["x"], prox["y"]==curr["y"],
                prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"],
                prox["pc"]==2)


    pc2 = And(curr["pc"]==2, prox["x"]==curr["x"]<<1, prox["y"]==curr["y"]>>1,
               prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"],
               prox["pc"]==0)


    pc3 = And(curr["pc"]==3, prox["x"]==curr["x"], prox["y"]==curr["y"],
               prox["m"]==curr["m"], prox["n"]==curr["n"], prox["r"]==curr["r"],
               prox["pc"]==curr["pc"], Not(curr["y"]>0))

    return Or(pc0_1, pc0_2, pc1_1, pc1_2, pc2, pc3)


def preCond(state):
    return And(state["m"]>=0, state["n"]>=0,state["y"]==state["n"], state["x"]==state["m"], state["r"]==0)

def posCond (state):
    return (state["r"] == state["m"]*state["n"])

def b (state):
    return (state["y"] > 0)

def unfold(declare, pre,trans, b, pos, n): # n = número de bits da variável y
    n = 3 * n # 3 unfolds ao ciclo
    state = {i: declare(i) for i in range(n)}
```

```
    s = Solver()
    s.add(pre(state[0]))
    for i in range(n-1):
        if i % 3 == 0:
            s.add(b(state[i]))
        s.add(trans(state[i], state[i+1]))
    s.add(Not(posCond(state[n-1])))

    if s.check() == sat:
        print("O programa não está correto.")
        m = solver.model()

        for v in state[0]:
            print(v, "=", m[state[0][v]])
    else:
        print("O programa está correto.")

N = 16
unfold(declare, preCond,trans, b, posCond, N)
```

```
O programa está correto.
```

2. Pretende-se verificar a correção total deste programa usando a metodologia dos invariantes e a metodologia do "single assignment unfolding". Para isso,

    1. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.
    2. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.
    3. Construa a definição iterativa do "single assignment unfolding" usando um parâmetro limite $$,N,$$ e aumentando a pré-condição com a condição

       ```
       $$(n < N)\,\land\,(m<N)$$
       ```

       O número de iterações vai ser controlado por este parâmetro $$N$$

✓ 0 s concluído à(s) 23:20