#TP3

Paulo Freitas - A100053

Pedro Santos - A100110

Enunciado

Considere-se de novo o algoritmo estendido de Euclides apresentado no TP2 mas usando o tipo dos inteiros e um parâmetro

$$N > 0$$

```
INPUT  a, b : Int
assume  a > 0 and b > 0 and a < N and b < N
r, r', s, s', t, t' = a, b, 1, 0, 0, 1
while r' != 0
  q = r div r'
  r, r', s, s', t, t' = r', r − q × r', s', s − q × s', t', t − q × t'
OUTPUT r, s, t
```

Exercício 2

Este exercício é dirigido à prova de correção do algoritmo estendido de Euclides

```
1. Construa a asserção lógica que representa a pós-condição do
algoritmo. Note que a definição da função gcd é gcd(a,b) = min{r > 0 |
E [s,t] . r = a * s + b * t}.
2. Usando a metodologia do comando havoc para o ciclo, escreva o
programa na linguagem dos comandos anotados (LPA). Codifique a pós-
condição do algoritmo com um comando assert .
3. Construa codificações do programa LPA através de transformadores de
predicados: “weakest pre-condition” e “strongest post-condition”.
4. Prove a correção  do programa LPA em ambas as codificações.
```

%%capture !yes | pip install pysmt !apt-get install libgmp3-dev !yes | pysmt-install --z3 --msat

file = '/usr/local/lib/python3.10/dist-packages/pysmt/smtlib/parser/**init**.py' with open(file, 'r') as f: code = f.read() new_code = code.replace('USE_CYTHON = True', 'USE_CYTHON = False')

with open(file, 'w') as f: f.write(new_code)

```
from pysmt.shortcuts import *
from pysmt.typing import INT,BVType
from pysmt.logics import QF_NIA
import itertools
```

EXERCÍCIO 2

Exercício 2.1

```
a = Symbol('a',INT)
b = Symbol('b',INT)
r = Symbol('r',INT)

n = Symbol('n',INT)
r_ = Symbol('r_',INT)
s = Symbol('s',INT)
s_ = Symbol('s_',INT)
t = Symbol('t',INT)
t_ = Symbol('t_',INT)
q = Symbol('q' ,INT)

inv = And(GT(r,Int(0)), LT(r,n), Equals(r, Plus(Times(a,s),
Times(b,t))))

pos = And(Not(And(LE(r_, r), Equals(Plus(Times(a, s_), Times(b, t_)),
r_))), inv)

print(pos)
```

```
((! ((r_ <= r) & ((... + ...) = r_))) & ((0 < r) & (r < n) & (r =
((... * ...) + (... * ...)))))
```

Exercício 2.2

Usando a metodologia do comando havoc para o ciclo, escreva o programa na linguagem dos comandos anotados (LPA). Codifique a pós-condição do algoritmo com um comando assert .

Auxiliar

A função *prove* verifica a validade de uma certa formula lógica, com recurso a um solver.

```
def prove(f):
    with Solver(name="z3") as solver:
        solver.add_assertion(Not(f))
        if solver.solve():
            print("Proved")
        else:
            print("Failed to prove")
```

LPA

Primeiro fazemos o ciclo do programa extentido de euclides em LPA. Ou seja:

w ≡ {assume b; S; W} || {^assume b}

S ≡ q = r div r'; r = r'; r' = r - q * r'; s = s'; s' = s - q * s'; t = t'; t' = t - q * t'
w ≡ {assume r' != 0; q = r div r'; r = r'; r' = r - q * r'; s = s'; s' = s - q * s'; t = t'; t' = t - q * t'; W} || {assume not r' != 0}
≡ {assume not r' == 0; q = r div r'; r = r'; r' = r - q * r'; s = s'; s' = s - q * s'; t = t'; t' = t - q * t'; W} || {assume r' == 0}

HAVOC

A partir do LPA temos o seguinte havoc:

```
func = Implies(And(Not(Equals(r, Int(0))),inv),substitute(
    substitute(
        substitute(
            substitute(
                substitute(
                    substitute(inv,
                               {t_:t - q * t}),
                    {t:t_}),
                {s_:s - q * s_}),
            {s:s_}),
        {r_:r - q * r_}),
    {r:r_}))

havoc = ForAll([r, r_, s, s_, t, t_], func)
axioms = And(havoc, And(Equals(r, a), Equals(r_, b), Equals(s,
Int(1)), Equals(s_, Int(0)), Equals(t, Int(0)), Equals(t_, Int(1))))

prove(havoc)
prove(axioms)

Proved
Proved
```

2.3.1 Construa codificações do programa LPA através de transformadores de predicados: “weakest pre-condition”

Neste código, procuramos mostrar que a pré condição é verdadeira, para assim ser verdadeira no fim de cada ciclo. Para isso usamos uma metodologia LPA. Ou seja:

[assume a > 0 and b > 0 and a < N and b < N; havoc f; r = a; r' = b; s = 1; s' = 0; t = 0; t' = 1;]

=

(a > 0 and b > 0 and a < N and b < N) -> [havoc x and ([r = a]) [r' = b]) [s = 1]) [s' = 0]) [t = 0]) [t' = 1])), pos]

```
WPC = substitute(
        substitute(
            substitute(
                substitute(
```

```
                    substitute(
                        substitute(pos,{t_:Int(1)}),
                        {t:Int(0)}),
                    {s_:Int(0)}),
                {s:Int(1)}),
            {r_:b}),
        {r:a})
```

2.3.2 Construa codificações do programa LPA através de transformadores de predicados: “strongest pos-condition”

A "strongest pos-condition" é a condição mais forte que, se verdadeira após a execução de um estado inicial satisfazendo uma pré-condição, garante que a pós-condição também seja satisfeita. Ou seja:

[assume a > 0 and b > 0 and a < N and b < N; havoc x; r = a; r' = b; s = 1; s' = 0; t = 0; t' = 1; pos]

= [assume a > 0 and b > 0 and a < N and b < N; havoc x; r = a; r' = b; s = 1; s' = 0; t = 0; t' = 1;] -> ((r = a * s + b * t and r > 0 and r < N), not(r_ <= r and r_= a * s_+ b * t_))

= (a > 0 and b > 0 and a < N and b < N and havoc x and r = a and r' = b and s = 1 and s' = 0 and t = 0 and t' = 1) -> ((r = a * s + b * t and r > 0 and r < N), not(r_ <= r and r_= a * s_+ b * t_)))

```
SPC = And(Equals(r, a),
          Equals(r_, b),
          Equals(s, Int(1)),
          Equals(s_, Int(0)),
          Equals(t, Int(0)),
          Equals(t_, Int(1)),
         )
```

2.4 Prove a correção do programa LPA em ambas as codificações.

Basta invocar a WPC e SPC para fazer a prova.

```
pre = (a > 0) and (b > 0) and (a < n) and (b < n)

WPC_final = Implies(pre,
                    And(WPC,
                       axioms,
                       pos))

SPC_final = Implies(And(pre,
                    axioms,
                    SPC),
                    pos)
```

```
print('WPC')
prove(WPC_final)
print('SPC')
prove(SPC_final)
```

```
WPC
Proved
SPC
Proved
```