Trabalho Prático 4 Grupo 24

Gabriel Antunes a101101 Guilherme Pinho a105533

Problema 2

Compreensão do problema

Antes de iniciar verifiquemos o comportamento do algoritmo

```python
from z3 import *

def euclides(a, b):
    r, rr, s, ss, t, tt = a, b, 1, 0, 0, 1
    print(f"Input:  a: {a}, b: {b}, r: {r}, r': {rr}, s: {s}, s': {ss}, t: {t}, t': {tt}")

    while rr != 0:
        q = r // rr
        r, rr, s, ss, t, tt = rr, r - q * rr, ss, s - q * ss, tt, t - q * tt
        print(f"While:  a: {a}, b: {b}, r: {r}, r': {rr}, s: {s}, s': {ss}, t: {t}, t': {tt}, q: {q}")

    print(f"Final:  a: {a}, b: {b}, r: {r}, r': {rr}, s: {s}, s': {ss}, t: {t}, t': {tt}")
    print(f"Result: r: {r}\n")
    return r

euclides(12,18)
euclides(13,27)
euclides(200,120)
```

```
Input:  a: 12, b: 18, r: 12, r': 18, s: 1, s': 0, t: 0, t': 1
While:  a: 12, b: 18, r: 18, r': 12, s: 0, s': 1, t: 1, t': 0, q: 0
While:  a: 12, b: 18, r: 12, r': 6, s: 1, s': -1, t: 0, t': 1, q: 1
While:  a: 12, b: 18, r: 6, r': 0, s: -1, s': 3, t: 1, t': -2, q: 2
Final:  a: 12, b: 18, r: 6, r': 0, s: -1, s': 3, t: 1, t': -2
Result: r: 6

Input:  a: 13, b: 27, r: 13, r': 27, s: 1, s': 0, t: 0, t': 1
While:  a: 13, b: 27, r: 27, r': 13, s: 0, s': 1, t: 1, t': 0, q: 0
While:  a: 13, b: 27, r: 13, r': 1, s: 1, s': -2, t: 0, t': 1, q: 2
While:  a: 13, b: 27, r: 1, r': 0, s: -2, s': 27, t: 1, t': -13, q: 13
Final:  a: 13, b: 27, r: 1, r': 0, s: -2, s': 27, t: 1, t': -13
Result: r: 1

Input:  a: 200, b: 120, r: 200, r': 120, s: 1, s': 0, t: 0, t': 1
```

```
While:  a: 200, b: 120, r: 120, r': 80, s: 0, s': 1, t: 1, t': -1, q:
1
While:  a: 200, b: 120, r: 80, r': 40, s: 1, s': -1, t: -1, t': 2, q:
1
While:  a: 200, b: 120, r: 40, r': 0, s: -1, s': 3, t: 2, t': -5, q: 2
Final:  a: 200, b: 120, r: 40, r': 0, s: -1, s': 3, t: 2, t': -5
Result: r: 40


40
```

Alínea a):

Construa a asserção lógica que representa a pós-condição do algoritmo. Note que a definição da função gcd é

$$gcd(a,b) \equiv min\{r > 0 \vee \exists s,t \quad r = a * s + b * t\}$$

Pós-Condição: $r = gcd(a,b) \wedge r = a \cdot s + b \cdot t \wedge r > 0$

Alínea b):

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** .

Na metodologia *havoc*, o ciclo (${\sf while} \; b \;{\sf do }\{\theta\} \;C$), com anotação de invariante θ é transformado num fluxo não iterativo da seguinte forma

$$ {{\sf assert}\; \theta\; ; \sf havoc }\;\vec{x} \; ; (\,({\sf assume }\; b \wedge \theta \; ; \; C \; ; {\sf assert}\;\theta \; ; {\sf assume}\; \mathit{False}) \: || \: {\sf assume}\; \neg b \wedge \theta \,) $$

Algoritmo de Euclides Estendido

```
INPUT: a, b
assume a > 0 and b > 0
# Pré-Condição

r, r', s, s', t, t' = a, b, 1, 0, 0, 1

while r' != 0
    { gcd(a, b) = gcd(r, r') ∧ r = a * s + b * t ∧ r' = a * s' + b *
t' }   # Invariante

    q = r div r'

    r, r', s, s', t, t' = r', r − q × r', s', s − q × s', t', t − q ×
t'
```

```
assert r = gcd(a, b) and r = a * s + b * t
# Pós-Condição
OUTPUT: r
```

Programa na Linguagem dos Comandos Anotados (LPA)

preCond = a > 0 and b > 0

inv = gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' + b * t'

posCond = r = gcd(a, b) and r = a * s + b * t

C = q = r div r'; r, r', s, s', t, t' = r', r − q × r', s', s − q × s', t', t − q × t'

xVect = r, r', s, s', t, t'

[assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; ((assume r != 0 and inv; C; assume False) || assume r == 0 and inv); assert posCond]

Alínea c):

Construa codificações do programa LPA através de transformadores de predicados “strongest post-condition” e prove a correção do programa LPA.

Antes de desenvolver o programa em LPa através de transformafores de predicados SPC, recordemos os mesmos:

Neste caso, a denotação `[C]` associa a cada fluxo `C` um predicado que caracteriza a sua correcção em termos lógicos (a sua VC) segundo a técnica SPC, sendo calculada pelas seguintes regras.

$ \begin{array}{l} [{\sf skip}] = True \ [{\sf assume}:\phi] = \phi \ [{\sf assert}:\phi] = \phi \ [x = e] = (x = e)\ [(C_1 || C_2)] = [C_1] \vee [C_2] \ \ [C , ; {\sf skip};] = [C] \ [C , ;{\sf assume}:\phi] = [C] \ wedge \phi \ [C , ;{\sf assert}:\phi] = [C] \to \phi \ [C , ; x = e] = [C] \wedge (x = e)\ [C,; (C_1 || C_2)] = [(C ; C_1) || (C; C_2)] \end{array} $

[assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; ((assume r != 0 and inv; C; assume False) || assume r == 0 and inv); assert posCond]

=

[assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; ((assume r != 0 and inv; C; assume False) || assume r == 0 and inv)] -> posCond

=

[(assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; (assume r != 0 and inv; C; assume False)) || (assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; assume r == 0 and inv)] -> posCond

=

[(assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; (assume r != 0 and inv; C; assume False))] or [(assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; assume r == 0 and inv)] -> posCond

=

[assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; (assume r != 0 and inv; C; assume False)] or [assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect; assume r == 0 and inv] -> posCond

=

([assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect] and (assume r != 0 and inv; C; assume False) or [assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv; havoc xVect] and r == 0 and inv) -> posCond

=

([assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv] and ForAll xVect. (assume r != 0 and inv; C; assume False) or [assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1; assert inv] and ForAll xVect. r == 0 and inv) -> posCond

=

([assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1] -> inv and (ForAll xVect. (assume r != 0 and inv; C; assume False)) or [assume preCond; r, r', s, s', t, t' = a, b, 1, 0, 0, 1] -> inv and (ForAll xVect. r == 0 and inv)) -> posCond

=

([assume preCond] and (r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. (assume r != 0 and inv; C; assume False)) or [assume preCond] and (r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. r == 0 and inv)) -> posCond

=

((preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. (assume r != 0 and inv; C; assume False)) or (preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. r == 0 and inv)) -> posCond

=

((preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. ((r != 0 and inv) and [C] and False)) or (preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv and (ForAll xVect. r == 0 and inv)) -> posCond

= Simplificando

((preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> inv) and ((ForAll xVect. ((r != 0 and inv) and [C] and False) or (r == 0 and inv))) -> posCond

= Note-se que ((r != 0 and inv) and [C] and False) = False

= Note-se também que (False or C) = C

((preCond and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> (inv and (ForAll xVect. (r == 0 and inv)))) -> posCond

= Por fim, substituindo:

(((a > 0 and b > 0) and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> (gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' + b * t')) and

(ForAll r, r', s, s', t, t'. (r == 0 and (gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' + b * t'))) -> r = gcd(a, b) and r = a * s + b * t

Uma vez que já temos a expressão:

(((a > 0 and b > 0) and r, r', s, s', t, t' = a, b, 1, 0, 0, 1) -> (gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' + b * t')) and

(ForAll r, r', s, s', t, t'. (r == 0 and (gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' + b * t'))) -> r = gcd(a, b) and r = a * s + b * t

Passemos à prova:

```python
def prove(f):

    with Solver() as s:
        s.add(Not(f))
        if s.check() == sat:
            print("Failed to prove.")
        else:
            print("Proved.")

a = Int('a')
b = Int('b')
r = Int('r')
rr = Int('rr')
s = Int('s')
ss = Int('ss')
t = Int('t')
tt = Int('tt')
q = Int('q')

# Definir gcd como uma função
gcd = Function('gcd', IntSort(), IntSort(), IntSort())
gcdAx1 = ForAll([a], Implies(a >= 0, gcd(a, 0) == a))
gcdAx2 = ForAll([b], Implies(b >= 0, gcd(0, b) == b))
gcdAx3 = ForAll([a, b], Implies(And(a > 0, b > 0), gcd(a, b) == gcd(b,
```

```
a % b)))

gcdF = And(gcdAx1, gcdAx2, gcdAx3)

preCond = And(a > 0, b > 0)

axioms = And(
    r == a,
    rr == b,
    s == 1,
    ss == 0,
    t == 0,
    tt == 1
)

pre = And(preCond, axioms)

# inv = gcd(a, b) = gcd(r, r') and r = a * s + b * t and r' = a * s' +
b * t'
inv = And(
    gcd(a, b) == gcd(r, rr),
    r == a * s + b * t,
    rr == a * ss + b * tt
)

preFinal = (Implies(pre, inv))

exitCond = ForAll([r,rr,s,ss,t,tt], And((r == 0), inv))

# posCond = r = gcd(a, b) and r = a * s + b * t
posCond = And(
    r == gcd(a, b),
    r == a * s + b * t
)

posFinal = Implies(exitCond, posCond)

spc = And(preFinal, posFinal)

prove(Implies(gcdF, spc))
```

```
Proved.
```