Trabalho 4

Pedro Gomes a91647
Francisco Teófilo a93741

In [29]:

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

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

In [30]:

```
from z3 import *
import sys, os
import matplotlib.pyplot as plt
```

Considerando o programa para multiplicação de dois inteiros de precisão limitada a 16 bits.
Programa:

```
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. Queremos provar por indução a sua terminação.

Realizamos a prova por indução com o `lookahead`.
Definimos o variante do programa como:

$$V(\alpha_i) \equiv y_i$$

Utilizando indução com um lookahead de l, queremos então provar, para um dado traço $\alpha = \{\alpha_i \mid i = 0, 1, \ldots, k-1\}$ **de um FOTS, que o programa termina (ou seja, a variável** $\texttt{pc}$ **vai tomar o valor** 3**).**

Este traço é gerado da seguinte forma:

$$\alpha \equiv \texttt{init}(\alpha_0) \ \wedge \ \forall_{i \in \{0, 1, \ldots, k-2\}}, \texttt{trans}(\alpha_i, \alpha_{i+1})$$

Para que se verifique isto, as seguintes propriedades têm de ser verificadas:

- **Positivo:** $\forall_i.\ \alpha_i \in \alpha,\ V_i(\alpha) \geq 0$
- **Decrescente:** $\forall_i.\ \alpha_i \in \left(\alpha \setminus \alpha_{k-1}\right),\ V(\alpha_{i+l}) < V(\alpha_i)$
- **Útil:** $V(\alpha_i) = 0 \rightarrow \left(\texttt{pc}_{i+l} = 3\right)$

Implementação indução com lookahead:

In [31]:

```
def inducao_always(declare, init, trans, var, prop, l, bits=16):
    # Declarar o traço
    solver = Solver()
    traco = {i: declare(i, bits) for i in range(2)}

    # Testar caso de base
    solver.add(init(traco[0]))
    solver.add(Not(var(traco[0], trans, l)))

    if solver.check() == sat:
        print("inducao breaks down no traco  inicial.")
        m = solver.model()

        for v in traco[0]:
            print(v, "=", m[traco[0][v]])
        return
    elif solver.check() != unsat:
        return

    # Testar caso indutivo
    solver = Solver()
    solver.add(var(traco[0], trans, l))
    solver.add(Not(var(traco[0], trans, l)))

    if solver.check() == sat:
        print("inducao breaks down no traco indutivo.")
        m = solver.model()

        for v in traco[0]:
            print(v, "=", m[traco[0][v]])
        return
    elif solver.check() == unsat:
        print(f"A propriedade \"{prop}\" é válida.")
```

Definimos então a geração do traço da seguinte forma:

$\texttt{init}(\alpha_i) \ \equiv \ \texttt{pc}_i = 0 \ \wedge \ r_i = 0 \ \wedge \ m_i \geq 0 \ \wedge \ n_i \geq 0 \ \wedge \ x_i = m_i \ \wedge \ y_i = n_i$

$$\texttt{trans}_0(\alpha_i, \alpha_{i+1}) \ \equiv \ \left[\texttt{pc}_i = 0 \ \wedge \ y_i > 0 \ \wedge \ x_{i+1} = x \ \wedge \ y_{i+1} = y \ \wedge \ m_{i+1} = m_i \ \wedge \ n_{i+1} = n_i \ \wedge \ r_{i+1} = r_i \ \wedge \ \texttt{pc}_{i+1} = 1\right]$$

$$\vee \ \left[\texttt{pc}_i = 0 \ \wedge \ y_i \leq 0 \ \wedge \ x_{i+1} = x \ \wedge \ y_{i+1} = y \ \wedge \ m_{i+1} = m_i \ \wedge \ n_{i+1} = n_i \ \wedge \ r_{i+1} = r_i \ \wedge \ \texttt{pc}_{i+1} = 3\right]$$

$$\mathtt{trans}_1(\alpha_i, \alpha_{i+1}) \equiv \left[\mathtt{pc}_i = 1 \land y_i \& 1 = 1 \land x_{i+1} = x \land y_{i+1} = y_i - 1 \land m_{i+1} = m_i \land n_{i+1} = n_i \land r_{i+1} = r_i + x_i \land \mathtt{pc}_{i+1} = 2\right]$$

$$\lor \left[\mathtt{pc}_i = 1 \land y_i \& 1 eq 1 \land x_{i+1} = x \land y_{i+1} = y_i \land m_{i+1} = m_i \land n_{i+1} = n_i \land r_{i+1} = r_i \land \mathtt{pc}_{i+1} = 2\right]$$

$\mathtt{trans}_2(\alpha_i, \alpha_{i+1}) \equiv \mathtt{pc}_i = 2 \land x_{i+1} = x_i \ll 1 \land y_{i+1} = y_i \gg 1 \land m_{i+1} = m_i \land n_{i+1} = n_i \land r_{i+1} = r_i \land \mathtt{pc}_{i+1} = 0$
$\mathtt{trans}_3(\alpha_i, \alpha_{i+1}) \equiv \mathtt{pc}_i = 3 \land x_{i+1} = x_i \land y_{i+1} = y_i \land m_{i+1} = m_i \land n_{i+1} = n_i \land r_{i+1} = r_i \land \mathtt{pc}_{i+1} = 3\ \mathtt{trans}(\alpha_i, \alpha_{i+1}) \equiv \mathtt{trans}_0(\alpha_i, \alpha_{i+1}) \land \mathtt{trans}_1(\alpha_i, \alpha_{i+1}) \land \mathtt{trans}_2(\alpha_i, \alpha_i + 1) \land \mathtt{trans}_3(\alpha_i, \alpha_{i+1})$

In [32]:

```python
def declare(i, bits=16):
    traco = {}
    traco["x"] = BitVec(f"x_{i}", bits)
    traco["y"] = BitVec(f"y_{i}", bits)
    traco["r"] = BitVec(f"r_{i}", bits)
    traco["m"] = BitVec(f"m_{i}", bits)
    traco["n"] = BitVec(f"n_{i}", bits)
    traco["pc"] = BitVec(f"pc_{i}", bits)

    return traco

def init(traco):
    r1 = And(traco["pc"]==0)
    r2 = And(traco["r"]==0, traco["m"]>=0, traco["n"]>=0, traco["x"]==traco["m"], traco["y"]==traco["n"])
    return And(r1, r2)

def trans(prev, curr):
    # Condições para pc == 0
    cond1_pc0 = And(prev["pc"]==0, prev["y"]>0, curr["x"]==prev["x"], curr["y"]==prev["y"],
                    curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"],
                    curr["pc"]==1)
    cond2_pc0 = And(prev["pc"]==0, Not(prev["y"]>0), curr["x"]==prev["x"], curr["y"]==prev["y"],
                    curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"],
                    curr["pc"]==3)
    cond_pc0 = Or(cond1_pc0, cond2_pc0)

    # Condições para pc == 1
    cond1_pc1 = And(prev["pc"]==1, prev["y"]&1==1, curr["x"]==prev["x"], curr["y"]==prev["y"]-1,
                    curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"]+prev["x"],
                    curr["pc"]==2)
    cond2_pc1 = And(prev["pc"]==1, Not(prev["y"]&1==1), curr["x"]==prev["x"], curr["y"]==prev["y"],
                    curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"],
                    curr["pc"]==2)
    cond_pc1 = Or(cond1_pc1, cond2_pc1)

    # Condições para pc == 2
    cond_pc2 = And(prev["pc"]==2, curr["x"]==prev["x"]<<1, curr["y"]==prev["y"]>>1,
                   curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"],
                   curr["pc"]==0)

    # Condições para pc == 3
    cond_pc3 = And(prev["pc"]==3, curr["x"]==prev["x"], curr["y"]==prev["y"],
                   curr["m"]==prev["m"], curr["n"]==prev["n"], curr["r"]==prev["r"],
                   curr["pc"]==prev["pc"], Not(prev["y"]>0))

    return Or(cond_pc0, cond_pc1, cond_pc2, cond_pc3)
```

In [33]:

```python
def variant(traco):
    return traco["y"]

def var_positivo(traco, trans, l=3):
    tracos = {i: declare(-i) for i in range(1, l+1)}
    c1 = And([trans(tracos[i], tracos[i+1]) for i in range(1, l)] + [trans(traco, tracos[1])])
    c2 = variant(tracos[l])>=0
    r = ForAll(list(tracos[l].values()), Implies(c1, c2))
    return r

def var_decrescente(traco, trans, l=3):
    tracos = {i: declare(-i) for i in range(1, l+1)}
    c1 = And([trans(tracos[i], tracos[i+1]) for i in range(1, l)] + [trans(traco, tracos[1])])
    c2 = Or(variant(tracos[l])<variant(traco), variant(tracos[l])==0)
    r = ForAll(list(tracos[l].values()), Implies(c1, c2))
    return r

def var_util(traco, trans, l):
    tracos = {i: declare(-i) for i in range(1, l+1)}
    c1 = And([trans(tracos[i], tracos[i+1]) for i in range(1, l)] + [trans(traco, tracos[1])])
    c2 = Implies(variant(tracos[l])==0, tracos[l]["pc"]==3)
    r = ForAll(list(tracos[l].values()), Implies(c1, c2))
    return r
```

Prova por Indução com Lookahead:

In [34]:

```python
bits = 16
inducao_always(declare, init, trans, var_positivo, "positivo", 1, bits)
inducao_always(declare, init, trans, var_decrescente, "decrescente", 3, bits)
inducao_always(declare, init, trans, var_util, "útil", 4, bits)
```

```
A propriedade "positivo" é válida.
A propriedade "decrescente" é válida.
A propriedade "útil" é válida.
```

2. Correção Total

a) Forma recursiva deste programa:

```
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
```

Fazemos a desdobragem 2 vezes:

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
if(y > 0): (1)
  if(y & 1 == 1): (2)
    y1,r1 = y-1, x
  else:
    y1,r1 = y, r;
  x1, y2 = x<<1, y1>>1

  if(y2 > 0): (3)
    if(y2 & 1 == 1): (4)
      y3,r2 = y2-1, r1*x1;
    else:
      y3, r2 = y2, r1;
      x2, y4 = x1 << 1, y3 >>1
      assert not(y4>0);
  else:
      x2, y4, r2 == x1, y2, r1
else:
x2, y4, r2 == x, y, r
assert r2 == n*m
```

Converter para os diferentes fluxos:

```
#Não entra em (1)
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume not(y>0); x2, y4, r2 == x, y, r

#Entra em todos
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume(y & 1 == 1); y1,r1 = y-1, x;x1, y2 = x<<1, y1>>1;
assume(y2>0);
assume(y2 & 1 == 1);y3,r2 = y2-1, r1*x1;x2, y4 = x1 << 1, y3 >>1;assert not(y4>0);

#Entra em todos exceto (4)
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume(y & 1 == 1); y1,r1 = y-1, x;x1, y2 = x<<1, y1>>1;
assume(y2>0);
assume not(y2 & 1 == 1);y3,r2 = y2,r1;x2, y4 = x1 << 1, y3 >>1;assert not(y4>0);

# Não entra em (3)
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume(y & 1 == 1); y1,r1 = y-1, x;x1, y2 = x<<1, y1>>1;
assume not(y2>0); x2, y4, r2 == x1, y2, r1

#Entra em (1) mas falha em (2) e (3)
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume not(y & 1 == 1); y1,r1 = y, r; x1, y2 = x<<1, y1>>1;
assume not(y2>0);x2, y4, r2 == x1, y2, r1

#Entra em (1),falha em (2), mas entra em (3) e (4)
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume not(y & 1 == 1); y1,r1 = y, r; x1, y2 = x<<1, y1>>1;
assume(y2>0);
assume(y2 & 1 == 1);y3,r2 = y2-1, r1*x1;x2, y4 = x1 << 1, y3 >>1;assert not(y4>0);

#Entra em (1),falha em (2),entra em (3) e falha (4)

assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assume (y>0);
assume not(y & 1 == 1); y1,r1 = y, r ; x1, y2 = x<<1, y1>>1;
assume(y2>0);
assume not(y2 & 1 == 1);y3, r2 = y2, r1;x2, y4 = x1 << 1, y3 >>1;assert not(y4>0)
);
assert r2 == n*m
```

b)Invariante e correção

Começamos por definir as pré e pós condições, assim como o invariante do ciclo.

$[\mathrm{P}] \equiv \phi \rightarrow \theta \wedge \forall \vec{x}. ((b \wedge \theta \rightarrow [C;\text{assert } \theta]) \wedge (\neg b \wedge \theta \rightarrow \psi))$

$\phi \equiv m \geq 0 \wedge n \geq 0 \wedge x = m \wedge y = n \wedge r = 0$

$\theta \equiv y \geq 0 \wedge x \cdot y + r = m \cdot n$

$b \equiv y > 0$

$\psi \equiv r = m \cdot n$

$f \equiv y \& 1 = 1$

$[C;\text{assert } \theta] \equiv [(C_1 \,||\, C_2);\text{assert } \theta] = [C_1 ;\text{assert } \theta] \wedge [C_2 ;\text{assert } \theta] = \left(f \rightarrow \theta[x/x \ll 1][y/(y-1) \gg 1][r/r+x]\right) \wedge \left(\neg f \rightarrow \theta[x/x \ll 1][y/ \gg 1]\right)$

Procedemos à sua correção através do método `Havoc`.

In [35]:

```
def havoc(bits=16):
    m, n, r, x, y = BitVecs("m n r x y", bits)

    pre = And(m >= 0, n >= 0, r == 0, x == m, y == n)
    pos = r == m * n
    inv = And(y >= 0, x*y+r == m*n)
    b = y > 0
    if_cond = y & 1 == 1

    cycle1 = Implies(if_cond, substitute(substitute(substitute(inv, (x, x<<1)), (y, (y-1)>>1)), (r, r+x)))
    cycle2 = Implies(Not(if_cond), substitute(substitute(inv, (x, x<<1)), (y, y>>1)))

    start = inv
    cycle = ForAll([x, y, r], Implies(And(b, inv), And(cycle1, cycle2)))
    end = Implies(And(Not(b), inv), pos)

    prove(Implies(pre, And(start, cycle, end)))
```

In [37]:

```
bits = 16
havoc(bits)
```

```
proved
```

c) Definição iterativa com Unfold

Cada corpo do ciclo que é executado deste programa faz a variável y ser pelo menos dividida por dois $y' <= y/2$, logo, o programa termina após o maior valor que y pode tomar ser dividido vezes suficientes para ser lhe ser atribuído um valor inferior a 1. Seja N o número de vezes que o corpo do ciclo deve ser executado para terminar:

$$\frac{|y|_{\mathtt{maj}}}{2^{|N|_{\mathtt{min}}}} \leq 1 \Leftrightarrow 2^{|N|_{\mathtt{min}}} \geq |y|_{\mathtt{maj}}$$

Neste programa o maior valor que pode tomar é 2^{n-1}, sendo n o número de bits da variável. Logo:

$$2^{|N|_{\mathtt{min}}} \geq 2^{n-1} \Leftrightarrow |N|_{\mathtt{min}} \geq n-1,\ N \in Z \Rightarrow |N|_{\mathtt{min}} = n$$

Utilizando então a estratégia unfold, aproveitando a definição do FOTS acima utilizado, onde o traço deste irá conter a evolução das variáveis do programa:

$$\mathtt{unfold}(n) \equiv \phi[v/\alpha_0] \wedge \bigwedge_{i=0}^{3n-2} \mathtt{trans}(\alpha_i, \alpha_{i+1}) \wedge \bigwedge_{i=0}^{3n-1} b[v/\alpha_{3i}] \wedge \psi[v/\alpha_{3n-1}]$$

Então pode-se provar o `unfold` a negar na expressão anterior, e verificar que o resultado dessa expressão lógica é `unsat`.

Adicionamos ainda à pré-condição $(n < N) \wedge (m < N)$
, em que, o número de iterações éN
.

Implementação unfold

In [38]:

```
def pre(traco):
    r1 = And(traco["m"]>=0, traco["n"]>=0)
    r2 = And(traco["y"]==traco["n"], traco["x"]==traco["m"], traco["r"]==0)
    r3 = And(traco["n"]<N, traco["m"]<N)
    return And(r1, r2)

pos = lambda traco: traco["r"] == traco["m"]*traco["n"]
b = lambda traco: traco["y"] > 0

def bmc_unfold(declare, trans, pre, b, pos, n, bits=16):
    n = 3 * n
    traco = {i: declare(i, bits) for i in range(n)}
    solver = Solver()
    solver.add(pre(traco[0]))
    for i in range(n-1):
        if i % 3 == 0:
            solver.add(b(traco[i]))
        solver.add(trans(traco[i], traco[i+1]))
    solver.add(Not(pos(traco[n-1])))

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

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

Prova unfold

In [41]:

```
N, bits = 16, 16
bmc_unfold(declare, trans, pre, b, pos, N, bits)
```

```
O programa está correto.
```