Trabalho 4

Grupo 22 João Carlos Marques A84684

```
!pip install z3-solver
from z3 import *
```

```
Requirement already satisfied: z3-solver in c:\users\johny\anaconda3\lib\site-packages (4.8.13.0)
```

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

Assumindo uma variável program counter tal que quando o programa termina temos, $pc=3$, então temos de provar um propriedade de animação do tipo $F(G\phi)$ onde $\phi=(pc=3)$

Definindo o variante do programa como:

$$V(s)=y$$

Usaremos então indução por k - induction para provar que o programa termina, isto é que a variável pc consegue tomar o valor 3

Para tal ocorrer, teremos de verificar as seguintes propriedades do nosso variante:

- Não Negativo:

$$G(V(s)\geq 0)$$

- Decrescente:

$$G(\forall s'.trans(s,s')\Longrightarrow(V(s')<V(s)\lor V(s')=0))$$

- Útil:

$$G(V(s)=0\Longrightarrow\phi(s))$$

Implementação k - induction

```python
def kinduction_always(declare,init,trans,inv,k,p):
    s = Solver()
    state = {}
```

```python
    for i in range(k):
        state[i] = declare(i)
    s.add(init(state[0]))
    for i in range(k-1):
        s.add(trans(state[i],state[i+1]))
    s.add(Or([Not(inv(state[i])) for i in range(k)]))
    status = s.check()

    assert (status!=unknown)
    if (status==sat):
        print('Nao é verdade nos estados iniciais')
        m = s.model()
        print(i)
        for v in state[i]:
            print(v,'=',m[state[i][v]])
        return

    s = Solver()
    state = {}
    for i in range(k+1):
        state[i] = declare(i)
    for i in range(k):
        s.add(inv(state[i]))
        s.add(trans(state[i],state[i+1]))
    s.add(Not(inv(state[k])))
    status = s.check()

    assert (status!=unknown)
    if (status==sat):
        print('Nao é verdade nos estados iniciais')
        m = s.model()
        print(i)
        for v in state[i]:
            print(v,'=',m[state[i][v]])
        return
    print("A propriedade " + p + " é valida")
```

Derivando facilmente o estado inicial:

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

Podemos então definir as transições desta maneira:

$pc=0 \land pc'=1 \land y>0 \land x'=x \land y'=y \land n'=n \land m'=m \land r'=r$

$\lor$

$pc=0 \land pc'=3 \land y\leq 0 \land x'=x \land y'=y \land n'=n \land m'=m \land r'=r$

$\lor$

$pc = 1 \land pc' = 2 \land y & 1 = 1 \land x'= x \land y'=y -1 \land n'=n \land m'=m \land r' = r+x $

$\lor$

$pc=1 \land pc'=2 \land y \land 1 \neq 1 \land x'=x \land y'=y \land n'=n \land m'=m \land r'=r$

$\lor$

$pc=2 \land pc'=0 \land x'=x<\text{¿}1 \land y'=y>\text{¿}1 \land n'=n \land m'=m \land r'=r$

$\lor$

$pc=3 \land pc'=3 \land x'=x \land y'=y \land m'=m \land n'=n \land r'=r$

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

    return state

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"])

def trans(s,p):
    pc0_pc1 = And(s["pc"]==0, p["pc"]==1, s["y"]>0, p["x"]==s["x"],
p["y"]==s["y"], p["n"]==s["n"], p["m"]==s["m"], p["r"]==s["r"])
    pc0_pc3 = And(s["pc"]==0, p["pc"]==3, s["y"]<=0, p["x"]==s["x"],
p["y"]==s["y"], p["n"]==s["n"],p["m"]==s["m"], p["r"]==s["r"])
    pc1_pc2_1 = And(s["pc"]==1, p["pc"]==2, s["y"] & 1 == 1,
p["x"]==s["x"], p["y"]==s["y"]-1, p["n"]==s["n"], p["m"]==s["m"],
p["r"]==s["r"]+s["x"])
    pc1_pc2_2 = And(s["pc"]==1, p["pc"]==2, Not(s["y"] & 1 == 1),
p["x"]==s["x"], p["y"]==s["y"], p["n"]==s["n"], p["m"]==s["m"],
p["r"]==s["r"])
    pc2_pc0 = And(s["pc"]==2, p["pc"]==0, p["x"]==s["x"]<<1,
p["y"]==s["y"]>>1, p["n"]==s["n"], p["m"]==s["m"], p["r"]==s["r"])
    pc3_pc3 = And(s["pc"]==3, p["pc"]==3, p["x"]==s["x"],
p["y"]==s["y"], p["n"]==s["n"], p["m"]==s["m"], p["r"]==s["r"])

    return Or(pc0_pc1,pc0_pc3,pc1_pc2_1,pc1_pc2_2,pc2_pc0,pc3_pc3)

def naoNegativo(state):
    return (state['y']>=0)
```

```python
def decrescente(state):
    prox = declare(-1)
    zero = prox['y'] + 3 - state['pc'] == 0
    menor = prox['y']  < state['y']
    return (Implies(trans(state,prox), Or(zero,menor)))

def utilidade(state):
    return (Implies(state['y'] + 3 - state['pc'] == 0,
state['pc']==3))

kinduction_always(declare, init, trans, naoNegativo, 1, "\"não
negativo\"")

kinduction_always(declare, init, trans, decrescente, 7,
"\"decrescente\"")

kinduction_always(declare, init, trans, utilidade, 3, "\"útil\"")
```

```
A propriedade "não negativo" é valida
Nao é verdade nos estados iniciais
5
pc = 2
x = 15674
y = 0
m = 7837
n = 2
r = 15674
A propriedade "útil" é valida
```

1. Pretende-se verificar a correção total deste programa usando a metodologia dos invariantes e a metodologia do "single assignment unfolding". Para isso,
    a. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.
    b. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.
    c. Construa a definição iterativa do "single assignment unfolding" usando um parâmetro limite
    $$N$$
    d. e aumentando a pré-condição com a condição
    $$(n<N) \wedge (m<N)$$
    e. O número de iterações vai ser controlado por este parâmetro
    $$N$$

LPA

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

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

$b \equiv y > 0$

$c \equiv y \wedge 1 = 1$

$S \equiv (assume\, c; S1 || assume\, \neg c; S2)$

$W \equiv \{assume\, b; S; W\} || \{assume\, \neg b\}$

$H \equiv assume;\phi; W ; assert;\psi$

$inv \equiv y \geq 0 \wedge x * y + r = m * n$

$H^{¿} = \{I; H^{¿}\} || \{T\}$

$I \equiv assume\, b \wedge inv; S; assert\, inv$

$[I] \equiv ((y > 0) \land (y\ge 0 \land xy+r=mn) \to (c\to inv[x/x\gg1][y/(y-1)\ll1][r/r+x]) \land (\neg c \to inv[x/x\gg1][y/\ll1]))$

$\equiv (y > 0 \wedge x * y + r = m * n) \rightarrow (c \rightarrow inv[x/x \gg 1][y/(y-1) \ll 1][r/r+x]) \wedge (\neg c \rightarrow inv[x/x \gg 1][y/ \ll 1])¿$

$T \equiv assume\, \neg b \wedge inv \rightarrow \psi$

$[T] \equiv ((y \leq 0) \wedge (y \geq 0 \wedge x * y + r = m * n)) \rightarrow r = m * n$

$\equiv (y = 0 \wedge x * y + r = m * n) \rightarrow r = m * n$

Correção

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

    b = y>0
    inv = And(y>=0, x*y+r==m*n)
    c = y&1==1
    S1 = Implies(c, substitute(substitute(substitute(inv, (x, x<<1)),
(y, (y-1)>>1)), (r,r+x)))
    S2 = Implies(Not(c), substitute(substitute(inv, (x,x<<1)),
(y,y>>1)))
    pos = r == m*n

    I = Implies(And(b,inv), And(S1,S2))
    T = Implies(And(Not(b), inv), pos)

    prove(And(I,T))

correcao(8)
```

```
proved
```

Correção Havoc

$P \equiv assume\, \phi; assert\, inv; havoc\, \vec{x}; ¿$

$$[S; assert\, inv] \equiv (c \rightarrow inv[x/x \gg 1][y/(y-1) \ll 1][r/r+x]) \wedge (\neg c \rightarrow inv[x/x \gg 1][y/ \ll 1])$$

$$[P] \equiv \phi \rightarrow inv \wedge \forall \vec{x}.((b \wedge inv \rightarrow [S; assert\, inv]) \wedge (\neg b \wedge inv \rightarrow \psi))$$

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

    pre = And(m>=0, n>=0, r==0, x==m, y==n)
    b = y>0
    inv = And(y>=0, x*y+r==m*n)
    c = y&1==1
    S1 = Implies(c, substitute(substitute(substitute(inv, (x, x<<1)),
(y, (y-1)>>1)), (r,r+x)))
    S2 = Implies(Not(c), substitute(substitute(inv, (x,x<<1)),
(y,y>>1)))
    pos = r == m*n
    havoc=ForAll([x,y,r], Implies(And(b,inv),And(S1,S2)))
    fim = Implies(And(Not(b),inv),pos)

    prove(Implies(pre, And(inv,havoc,fim)))
```

```
correHavoc(8)
```

```
proved
```

SAU

Queremos então descobrir um parâmetro N tal que este corresponda ao valor máximo de iterações que o nosso ciclo pode executar antes de terminar. Ora como podemos observar a cada execução do ciclo a variável y será dividida por 2, sendo assim sabemos que o programa termina assim que y tomar um valor inferior a 1, então temos:

$$y \div 2^N < 1$$

logo

$$y < 2^N$$

O maior valor que y pode tomar será 2^{n-1}, onde n corresponde ao numero de bits da variável, sendo este um programa que faz a multiplicação de dois inteiros de precisão limitada a 16 bits então o maior valor que y pode tomar será 2^{16-1} logo $ 2^{16-1} < 2^N$. Como a cada execução do ciclo teremos 3 estados do programa então o numero de transições deste será $3n-2$

```
def newinit(state,N):
    return (And(And(state['pc']==0,state['m']>=0, state['n']>=0,
state['x']==state['m'], state['y']==state['n'],state['r']==0),
And(state['n']<N, state['m']<N)))
```

```python
def pos(state):
    return (state['r']==state['m']*state['n'])

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

def unfold(declare,bits,N):
    s = Solver()
    state = {}
    for i in range(bits):
        state[i] = declare(i)
    s.add(newinit(state[0],N))



    if s.check() == unsat:
        print("O programa está incorreto.")
        m = s.model()

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

unfold(declare,16,16)
```

```
O programa está correto.
```