

Lógica Computacional 21/22

Trabalho Prático 4

ID: Pergunta 1 - Grupo 1

Equipa:

1. Alef Keuffer (A91683)
2. Alexandre Rodrigues Baldé (A70373)

Preparação do código

Dependências

```
In [ ]: %pip install pysmt

from pysmt.shortcuts import *
from pysmt.typing import *

Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pysmt in /home/alexa_wsl/.local/lib/python3.10/site-packages (0.9.0)
Requirement already satisfied: six in /home/alexa_wsl/.local/lib/python3.10/site-packages (from pysmt) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

Enunciado do problema

Python Anotado

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
2. 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

e aumentando a pré-condição com a condição

$\$(n < N) \wedge \text{and}((m < N))$

O número de iterações vai ser controlado por este parâmetro

N

Pergunta 1

FOTS para o programa anotado

Construa-se um FOTS que modela o programa anotado do enunciado, de forma a utilizar-se k -indução para provar a sua terminação.

Considerem-se as variáveis x, y, r, m, n parte do FOTS, assim como uma variável pc que indica a que instrução em que a execução do programa está num dado momento.

Defina-se agora a função que declara as variáveis do FOTS:

Variáveis e estado inicial do FOTS

```
In [ ]: word_len = 16

BV16 = BVType(word_len)

def declare(i):
    state = {}

    state['x'] = Symbol('x'+str(i), BV16)
    state['y'] = Symbol('y'+str(i), BV16)
    state['r'] = Symbol('r'+str(i), BV16)
    state['m'] = Symbol('m'+str(i), BV16)
    state['n'] = Symbol('n'+str(i), BV16)
    state['pc'] = Symbol('pc'+str(i), INT)

    return state
```

Funções auxiliares

O PySMT não permite comparação entre vetores de bits e inteiros, logo a função abaixo é necessária para o fazer.

```
In [ ]: def bitVecIntComp(vec, i, bin_op, size = word_len):
    intVec = BV(i, size)
    if i == 0:
        intVec = BVZero(size)
    elif i == 1:
        intVec = BVOne(size)
    if bin_op == "EQ":
        return Equals(vec, intVec)
    elif bin_op == "NEQ":
        return Equals(vec, intVec)
    elif bin_op == "GT":
        return GT(BVToNatural(vec), Int(i))
    elif bin_op == "GE":
        return GE(BVToNatural(vec), Int(i))
    elif bin_op == "LT":
        return LT(BVToNatural(vec), Int(i))
    elif bin_op == "LE":
        return LE(BVToNatural(vec), Int(i))
    raise ValueError("Wrong operator supplied to helper function bitVecIntC
```

O predicado que define o estado inicial do FOTS é

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

Este predicado deriva da pre-condição do programa.

Note-se que, para resolver este problema, se consideram $m, n \geq 0$, ou seja, está-se a trabalhar no anel $\mathbb{Z}_{2^{16}}$, e havendo overflow, interpreta-se o resultado dessa forma.

```
In [ ]: def init(state):
    l = []

    l.append(bitVecIntComp(state['m'], 0, "GE"))
    l.append(bitVecIntComp(state['n'], 0, "GE"))
    l.append(bitVecIntComp(state['r'], 0, "EQ"))
    l.append(Equals(state['x'], state['m']))
    l.append(Equals(state['y'], state['n']))
    l.append(Equals(state['pc'], Int(0)))

    return And(l)
```

Função de transição

$$pc = 0$$

Quando a variável pc tem o valor 0, o programa está no ciclo, e há duas possibilidades:

- permanecer dentro do ciclo caso a condição seja verdadeira, ou
- transitar para o fim do programa caso a condição seja falsa.

Isto corresponde aos predicados:

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

e

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

$pc = 1$

Quando o valor de pc é 1, entrou-se dentro do ciclo.\ A próxima instrução é um `if_then_else` , donde:

- a condição é verdadeira e as variáveis y e r são alteradas, ou então
- a condição é falsa e as variáveis y e r não são alteradas.

Daqui, os seguintes predicados:

*

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

e

*

$$pc = 1 \wedge \neg(y \& 1 = 1) \wedge y' = y \wedge m' = m \wedge r' = r \wedge x' = x \wedge n' = n \wedge pc' = 2$$

$pc = 2$

Quando $pc = 2$, termina-se a execução do corpo do ciclo, e volta-se a $pc = 0$, para re-executar o ciclo se a condição se verificar.

$$pc = 2 \wedge y' = y >> 1 \wedge m' = m \wedge r' = r \wedge x' = x << 1 \wedge n' = n \wedge pc' = 0$$

$pc = 3$

Este caso corresponde à transição do estado final para ele próprio:

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

Note-se o uso da função `bitVecIntComp` para fazer as comparações entre `INT`s de PySMT e `BVType(16)` .

```
In [ ]:

def trans(curr, prox):
    l = []

    state1 = [Equals(curr['pc'], Int(0)),
              bitVecIntComp(curr['y'], 0, "GT"),
              Equals(prox['y'], curr['y']),
              Equals(prox['m'], curr['m']),
              Equals(prox['r'], curr['r']),
              Equals(prox['x'], curr['x']),
              Equals(prox['n'], curr['n']),
              Equals(prox['pc'], Int(1))]
    l.append(And(state1))

    state2 = [Equals(curr['pc'], Int(0)),
              bitVecIntComp(curr['y'], 0, "LE"),
              Equals(prox['y'], curr['y']),
              Equals(prox['m'], curr['m']),
              Equals(prox['r'], curr['r']),
              Equals(prox['x'], curr['x']),
              Equals(prox['n'], curr['n']),
              Equals(prox['pc'], Int(3))]
    l.append(And(state2))

    state3 = [Equals(curr['pc'], Int(1)),
              Equals(BVAnd(curr['y'], BVOne(word_len)), BVOne(word_len)),
              Equals(prox['y'], BVSub(curr['y'], BVOne(word_len))),
              Equals(prox['r'], BVAAdd(curr['r'], curr['x'])),
              Equals(prox['x'], curr['x']),
              Equals(prox['m'], curr['m']),
              Equals(prox['n'], curr['n']),
              Equals(prox['pc'], Int(2))]
    l.append(And(state3))

    state4 = [Equals(curr['pc'], Int(1)),
              NotEquals(BVAnd(curr['y'], BVOne(word_len)), BVOne(word_len))
              Equals(prox['y'], curr['y']),
              Equals(prox['r'], curr['r']),
              Equals(prox['x'], curr['x']),
              Equals(prox['m'], curr['m']),
              Equals(prox['n'], curr['n']),
              Equals(prox['pc'], Int(2))]
    l.append(And(state4))

    state5 = [Equals(curr['pc'], Int(2)),
              Equals(prox['x'], BVLShl(curr['x'], BVOne(word_len))),
              Equals(prox['y'], BVLShr(curr['y'], BVOne(word_len))),
              Equals(prox['m'], curr['m']),
              Equals(prox['n'], curr['n']),
              Equals(prox['r'], curr['r']),
              Equals(prox['pc'], Int(0))]
    l.append(And(state5))

    state6 = [Equals(curr['pc'], Int(3)),
              Equals(prox['y'], curr['y']),
              Equals(prox['m'], curr['m']),
              Equals(prox['r'], curr['r']),
              Equals(prox['x'], curr['x']),
              Equals(prox['n'], curr['n']),
              Equals(prox['pc'], Int(3))]
    l.append(And(state6))

return Or(l)


```

Geração de traço de execução

Como se viu nas aulas Teórico-Práticas, a função abaixo gera um traço de execução do FOTS.

```
In [ ]:  
def gera_traco(declare, init, trans, k):  
    trace = [declare(i) for i in range(k)]  
    s = Solver(name = 'z3')  
    s.reset_assertions()  
  
    s.add_assertion(init(trace[0]))  
  
    for i in range(k-1):  
        s.add_assertion(trans(trace[i], trace[i + 1]))  
  
    r = s.solve()  
    if r: # s is SAT  
        m = s.get_model()  
        for i in range(k):  
            print(f"state {i}:")  
            for v in trace[i]:  
                print(v, ' = ', s.get_value(trace[i][v]))  
    return  
  
    print('UNSAT!')  
return  
  
gera_traco(declare, init, trans, 20)
```

```
state 0:  
x = 0_16  
y = 1_16  
r = 0_16  
m = 0_16  
n = 1_16  
pc = 0  
state 1:  
x = 0_16  
y = 1_16  
r = 0_16  
m = 0_16  
n = 1_16  
pc = 1  
state 2:  
x = 0_16  
y = 0_16  
r = 0_16  
m = 0_16  
n = 1_16  
pc = 2  
state 3:  
x = 0_16  
y = 0_16  
r = 0_16  
m = 0_16  
n = 1_16  
pc = 0  
state 4:  
x = 0_16  
y = 0_16
```

```
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 5:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 6:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 7:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 8:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 9:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 10:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 11:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 12:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 13:
x = 0_16
y = 0_16
```

```

r = 0_16
m = 0_16
n = 1_16
pc = 3
state 14:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 15:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 16:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 17:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 18:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16
pc = 3
state 19:
x = 0_16
y = 0_16
r = 0_16
m = 0_16
n = 1_16

```

Prova de terminação do programa

Utilize-se k -lookahead para provar a terminação do programa acima, que é uma propriedade da forma $F(G \phi)$, onde $\phi \equiv pc = 3$.

É necessário encontrar um variante V que satisfaça as seguintes condições, descritas na folha de exercícios 7 das aulas Teórica-Práticas:

1. O variante é positivo ou nulo: $G (V(s) \geq 0)$
2. O variante é estritamente decrescente, ou atinge o valor 0, ou seja,
 - $G (\forall s' . \text{trans}(s, s') \rightarrow (V(s') < V(s) \vee V(s') = 0))$
3. Quando o variante é 0 verifica-se necessariamente ϕ , ou seja, $G (V(s) = 0 \rightarrow \phi(s))$

Considerando-se o uso de k -lookahead, a 2º condição pode ser alargada, permitindo-se que

o variante decresça necessariamente de 3 em 3 transições, e não necessariamente em transições consecutivas.

Lookahead

O lookahead em causa é de 3, que é o número mínimo de instruções necessárias para que pc vá do [início ao fim do ciclo](#).

Utilize-se o seguinte variante:

$$V(s) \equiv (y_s - pc_s) + 3$$

Antes de proceder à prova de terminação com este variante, algumas notas sobre esta escolha:

1. pc_s , durante a execução do ciclo, é tal que $0 \leq pc_s \leq 2$
2. y_s , durante a execução do ciclo, é tal que $y_s > 0 \Leftrightarrow y_s \geq 1$
3. $y_s - pc_s$, durante a execução do ciclo, é tal que $y_s - pc_s \geq -1$
4. Conclui-se que, durante a execução do ciclo, $(y_s - pc_s) + 3 \geq 2 > 0$

Após a execução do ciclo:

- $pc_s = 3 \Leftrightarrow$ o ciclo terminou
- o ciclo terminou $\Rightarrow y_s \leq 0$

Tendo em conta que y sofre um *shift* para a esquerda de 1 bit em cada iteração do ciclo, tem-se necessariamente que na última iteração do ciclo $y_s = 1$, e após essa execução final do corpo do ciclo, y_s será então 0, donde

$$\begin{aligned} (y_s - pc_s) + 3 &= -pc_s + 3 \quad (y_s = 0) \\ &= -3 + 3 \quad (\text{terminou-se o ciclo, donde } pc_s = 3) \\ &= 0 \\ &\geq 0 \end{aligned}$$

Função para k -indução, e definição do variante:

```
In [ ]: def variante(state):  
    return Plus(Minus(BVToNatural(state['y']), state['pc']), Int(3))
```

```
In [ ]:
def kinduction_always(declare, init, trans, inv, k):
    trace = [declare(i) for i in range(k + 1)]
    with Portfolio(["z3"],
                  logic="QF_BV",
                  incremental=False,
                  generate_models=False) as s:
        s.reset_assertions()
        s.add_assertion(init(trace[0]))
        for i in range(k - 1):
            s.add_assertion(trans(trace[i], trace[i + 1]))

        l = [Not(inv(trace[i])) for i in range(k)]
        s.add_assertion(Or(l))

        r = s.solve()

        if r:
            print('A prova por k-indução falhou no caso base:')
            print('O estado do FOTS que causou a falha:')
            m = s.get_model()
            for i in range(k):
                print(i)
                for v in trace[i]:
                    print(v, ' = ', m[trace[i][v]])
            return

        s.reset_assertions()
        for i in range(k):
            s.add_assertion(trans(trace[i], trace[i + 1]))
            s.add_assertion(inv(trace[i]))
        s.add_assertion(Not(inv(trace[k])))

        r = s.solve()

        if r:
            print(f'Falhou no passo de {k}-indução!')
            m = s.get_model()
            for i in range(k):
                print(i)
                for v in trace[i]:
                    print(v, ' = ', m[trace[i][v]])
            return

    if not r:
        print(f'Verifica-se a propriedade {inv}')
        return
```

Observe-se o uso de `Portfolio` para a criação de um `solver` que pode usar várias bibliotecas SMT em simultâneo.

Permite também especificar explicitamente a lógica a ser usada por todos os *solvers*, neste caso `QF_BV`.

Primeiro, prove-se a propriedade 1.: V é sempre positivo:

```
In [ ]: def positivo(state):
    prop = GE(variante(state), Int(0))
    return prop

kinduction_always(declare, init, trans, positivo, 3)
```

Verifica-se a propriedade <function positivo at 0x7fa8dc102b90>

Agora, a segunda condição:

```
In [ ]: def decresce(state):
    state1 = declare(-1)
    state2 = declare(-2)
    state3 = declare(-3)
    varnames = list(state1.values()) + list(state2.values()) + list(state3.

    return ForAll(
        varnames,
        Implies(
            And(trans(state, state1),
                trans(state1, state2),
                trans(state2, state3)
            ),
            Or(LT(variante(state3), variante(state)),
                Equals(variante(state3), Int(0)))
        )
    )
)

kinduction_always(declare, init, trans, decresce, 3)
```

Verifica-se a propriedade <function decresce at 0x7fa8dc1a52d0>

Finalmente, a terceira condição: que o variante ser zero implica a chegada à última instrução:

```
In [ ]: def terminacao(state):
    return Implies(Equals(variante(state), Int(0)), Equals(state['pc'], Int

kinduction_always(declare, init, trans, terminacao, 4)
```

Verifica-se a propriedade <function terminacao at 0x7fa8dc1a5a20>

Pergunta 2

Alínea a: Forma recursiva do programa em LPA

Recorde-se o programa original:

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

Seja W o seguinte programa em linguagem de anotações.

$$W \equiv \{\text{assume } (y \geq 0); S; W\} \quad || \quad \{\text{assume } \neg(y \geq 0)\}$$

onde S é o fluxo que corresponde à tradução da instrução dentro do ciclo para a linguagem de anotações:

```
if (y & 1 == 1):
    y, r = y - 1, r + x;
    x = x << 1;
    y = y >> 1;
```

equivale a

```
({
    assume (y & 1 == 1);
    y = y - 1;
    r = r + x;
} ||
{
    assume not (y & 1 == 1);
    skip;
})
x = x << 1;
y = y >> 1;
```

Logo o triplo de Hoare que especifica o programa na sua forma recursiva é:

$$\phi \equiv \text{assume } m \geq 0 \text{ and } n \geq 0 \text{ and } r = 0 \text{ and } x = m \text{ and } y = n;$$

$$\varphi \equiv \text{assert } r = m * n;$$

$$H \equiv \{\text{assume } \phi; W; \text{assume } \varphi\};$$

onde W equivale ao seguinte programa:

```
{
    assume (y > 0);
    {
        assume (y & 1 == 1);
        y = y - 1;
        r = r + x;
    } ||
    {
        assume not (y & 1 == 1);
        skip;
    }
    x = x << 1;
    y = y >> 1;
    W
} ||
{
    assume not (y > 0);
}
```

Alínea b: Invariante de ciclo, codificação em SMT e

Prova da correcção

Considere-se o seguinte invariante:

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

Analise-se a ideia do programa para explicá-lo.

Dados m, n , o programa coloca em r o valor $m \cdot n$, e fá-lo da seguinte forma:

- Considere-se o número n na sua representação binária, $n = \sum_{i=0}^n b_i \cdot 2^i$, onde b_i tem o valor do i -ésimo bit de n .
- Logo, $m \cdot n = m \cdot \sum_{i=0}^n b_i \cdot 2^i = \sum_{i=0}^n m \cdot b_i \cdot 2^i$. Daí as instruções do ciclo

```
if y & 1 == 1:  
    y, r = y - 1, r + x  
    x, y = x << 1, y >> 1
```

Após cada iteração do ciclo x sofre um *shift* para a esquerda (duplicando o seu valor), e se a condição se verificar - o i -ésimo bit de y for 1 - acrescenta-se o valor de $m \cdot 2^i$ ao resultado r .

Veja-se o código em execução para exemplificar:

```
In [ ]: def mul(m, n):  
    r = 0  
    x = m  
    y = n  
    def prnt(): print(f"x: {x}; y: {y}; y (bin): {format(y, 'b')}; r: {r};  
    prnt()  
    while y > 0:  
        if y & 1 == 1:  
            y, r = y-1, r+x  
            x, y = x<<1, y>>1  
            prnt()  
    return r, m * n  
  
mul(7, 136)
```

x: 7; y: 136; y (bin): 10001000; r: 0; x*y + r = 952
x: 14; y: 68; y (bin): 1000100; r: 0; x*y + r = 952
x: 28; y: 34; y (bin): 100010; r: 0; x*y + r = 952
x: 56; y: 17; y (bin): 10001; r: 0; x*y + r = 952
x: 112; y: 8; y (bin): 1000; r: 56; x*y + r = 952
x: 224; y: 4; y (bin): 100; r: 56; x*y + r = 952
x: 448; y: 2; y (bin): 10; r: 56; x*y + r = 952
x: 896; y: 1; y (bin): 1; r: 56; x*y + r = 952
x: 1792; y: 0; y (bin): 0; r: 952; x*y + r = 952
(952, 952)

Out[]:

Note-se que:

1. Antes da primeira iteração do ciclo, pela pré condição do programa, $x = m$, $y = n$, e $r = 0$, pelo que ele é válido.
2. Após a execução do ciclo, como se viu no caso do variante, tem-se que $y = 0$, pelo que o invariante também se mantém.

Codificação das VCs em PySMT, e prova de validade

```
In [ ]:
x = Symbol('x', BV16)
y = Symbol('y', BV16)
r = Symbol('r', BV16)
m = Symbol('m', BV16)
n = Symbol('n', BV16)

def prove(f):
    with Portfolio(
        ["z3"],
        logic="QF_BV",
        incremental=False,
        generate_models=False) as s:

        s.reset_assertions()
        s.add_assertion(Not(f))
        res = s.solve()
        if not res:
            print("The formula is a tautology!")
        elif res:
            print(is_sat(Not(f)))
            print("The formula is invalid!")
            print("Failing state:")
            print(s.get_model())
            print(f"x : {s.get_value(x)}")
            print(f"y : {s.get_value(y)}")
            print(f"r : {s.get_value(r)}")
            print(f"m : {s.get_value(m)}")
            print(f"n : {s.get_value(n)}")
```

Das fichas práticas, recorde-se que as condições de verificação de um programa com ciclo são:

$$\frac{\{\phi\}\text{skip}\{\theta\} \quad \{\theta \wedge b\}C\{\theta\} \quad \{\theta \wedge \neg b\}\text{skip}\{\psi\}}{\{\phi\}\text{while } b \text{ do } C\{\psi\}}$$

Já se viu acima o corpo do ciclo traduzido para a linguagem de anotações na versão recursiva do programa. Agora, para poder provar a segunda regra, coloca-se as pré e pós-condições, que envolvem o invariante e condição de ciclo

```
assume (m * n = x * y + r and y >= 0) and (y > 0)
{
    assume (y & 1 == 1);
    y = y - 1;
    r = r + x;
}
|||
{
    assume not (y & 1 == 1);
    skip;
}
x = x << 1;
y = y >> 1;
assert (m * n = x * y + r and y >= 0)
```

Utilizando as seguintes regras WPC para a geração de VCs:

$\text{[skip]} = \text{True}$
 $\text{[assume } \phi\text{]} = \text{True}$
 $\text{[assert } \phi\text{]} = \phi$
 $\text{[}x = e\text{]} = \text{True}$
 $\text{[(}C_1\text{||}C_2\text{)]} = \text{[}C_1\text{]} \wedge \text{[}C_2\text{]}$

$\text{[skip ; } C\text{]} = \text{[}C\text{]}$
 $\text{[assume } \phi ; C\text{]} = \phi \rightarrow \text{[}C\text{]}$
 $\text{[assert } \phi ; C\text{]} = \phi \wedge \text{[}C\text{]}$

Tem-se:

$$(\theta \wedge (y > 0)) \rightarrow \text{[(}C_1 ; C\text{)]} \wedge \text{[(}C_2 ; C\text{)]}$$

onde $\theta \equiv (m \cdot n - x \cdot y + r) \wedge y \geq 0$ é o invariante descrito acima, C_1, C_2, C são os programas

```
assume (y & 1 == 1);
y = y - 1;
r = r + x;
```

e

```
assume not (y & 1 == 1);
skip;
x = x << 1;
y = y >> 1;
assert (m * n == x * y + r and y >= 0)
```

, respetivamente.

No caso de $\text{[(}C_1 ; C\text{)]}$, fica-se com a VC

```
[assume (y & 1 == 1);
y = y - 1;
r = r + x;
x = x << 1;
y = y >> 1;
assert (m * n == x * y + r and y >= 0)]

== (y & 1 == 1) -> [y = y - 1;
                      r = r + x;
                      x = x << 1;
                      y = y >> 1;
                      assert (m * n == x * y + r and y >= 0)]

== (y & 1 == 1) -> (m * n == x * y + r and y >= 0)[y - 1 / y][r + x /
r][x << 1 / x][y >> 1 / y]

== (y & 1 == 1) -> (m * n == (x << 1) * ((y >> 1) - 1) + (r + x) and ((y
>> 1) - 1) >= 0)
```

No caso de $\text{[(}C_2 ; C\text{)]}$, fica-se com a VC

```

[assume not (y & 1 == 1);
skip;
x = x << 1;
y = y >> 1;
assert (m * n = x * y + r and y >= 0)]

== (y & 1 != 1) -> [x = x << 1;
y = y >> 1;
assert (m * n = x * y + r and y >= 0)]

== (y & 1 != 1) -> (m * n == x * y + r and y >= 0)[x << 1 / x][y >> 1 /
y]

-- ((y & 1 != 1) -> (m * n == x * y + r and y >= 0)) / ((y >> 1) -> (y >= 0))

```

Juntando o anterior, tem-se que a VC de $(\theta \wedge (y > 0)) \rightarrow [(C_1; C)] \wedge [(C_2; C)]$ é

```

(m * n = x * y + r and y >= 0) and (y > 0) ->
(
    (y & 1 == 1) -> (m * n = (x << 1) * ((y >> 1) - 1) + (r + x) and
y >= 0) and
    (y & 1 != 1) -> (m * n = (x << 1) * (y >> 1) + r and y >= 0)
)

== (m * n = x * y + r and y > 0) ->
(
    (y & 1 == 1) -> (m * n = (x << 1) * ((y >> 1) - 1) + (r + x) and y
>= 0) and
    (y & 1 != 1) -> (m * n = (x << 1) * (y >> 1) + r and y >= 0)
)
```

```
In [ ]: cycle_condition = bitVecIntComp(y, 0, "GT")

invariant = And(Equals(BVMul(m, n), BVAAdd(BVMul(x, y), r)), bitVecIntComp(y, 0, "GT"))

pre_condition = And([
    bitVecIntComp(m, 0, "GE"),
    bitVecIntComp(n, 0, "GE"),
    bitVecIntComp(r, 0, "EQ"),
    Equals(x, m),
    Equals(y, n)
])

post_condition = Equals(r, BVMul(m, n))

cond1 = Implies(pre_condition, invariant)
```

```
In [ ]: prove(cond1)
```

The formula is a tautology!

```
In [ ]:
aux1 = invariant
aux1 = substitute(aux1, {y : BVSub(y, BVOne(word_len))})
aux1 = substitute(aux1, {r : BVAdd(r, x)})
aux1 = substitute(aux1, {y : BVLShr(y, BVOne(word_len))})
aux1 = substitute(aux1, {x : BVLShl(x, BVOne(word_len)))})

c1_then_c = Implies(Equals(BVAnd(y, BVOne(word_len)), BVOne(word_len)), aux1)

aux2 = invariant
aux2 = substitute(aux2, {x : BVLShl(x, BVOne(word_len))})
aux2 = substitute(aux2, {y : BVLShr(y, BVOne(word_len)))})

c2_then_c = Implies(NotEquals(BVAnd(y, BVOne(word_len)), BVOne(word_len)), aux2)

premissa_cond2 = And(Equals(BVMul(m, n), BVAdd(BVMul(x, y), r)), bitVecToInt(c))

cond2 = Implies(premissa_cond2, And(c1_then_c, c2_then_c))
```

```
In [ ]:
prove(cond2)
```

The formula is a tautology!

```
In [ ]:
cond3 = Implies(And(invariant, Not(cycle_condition)), post_condition)
```

```
In [ ]:
prove(cond3)
```

The formula is a tautology!

Alínea c: SAU com limite de iterações em ciclo

```
In [ ]:
def prime(v):
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())
def fresh(v):
    return FreshSymbol(typename=v.symbol_type(), template=v.symbol_name() + "_")

# A classe "Single Assignment Unfold"
class SAU(object):
    """Trivial representation of a while cycle and its unfolding."""
    def __init__(self, variables, pre, pos, control, trans, sname="z3"):

        self.variables = variables           # variables
        self.pre = pre                         # pre-condition as a predicate in
        self.pos = pos                         # pos-condition as a predicate in
        self.control = control                 # cycle control as a predicate in
        self.trans = trans                     # cycle body as a binary transition
                                                # in "variables" and "prime variables"

        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [And([Not(control), pos])]

        # inicializa com uma só frame: a da terminação do ciclo

        self.solver = Solver(name=sname)

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        b = self.control
        S = self.trans.substitute(dict(zip(self.prime_variables, freshs)))
        W = self.frames[-1].substitute(dict(zip(self.variables, freshs)))

        self.frames.append(And([b, ForAll(freshs, Implies(S, W))]))

    def unfold(self, bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: número de tentativas ultrapassa o limite %d"
                      % bound)
                break

            f = Or(self.frames)
            if self.solver.solve([self.pre, Not(f)]):
                self.new_frame()
                n += 1
            else:
                print("sucesso na tentativa %d" % n)
                break
```

```
In [ ]:
N = 16384
zero = Int(0)
one = BVOne(word_len)

# O ciclo
x = Symbol('x', BV16)
y = Symbol('y', BV16)
r = Symbol('r', BV16)
m = Symbol('m', BV16)
n = Symbol('n', BV16)
variables = [x, y, r]

pre_condition = And([
    bitVecIntComp(m, 0, "GE"),
    bitVecIntComp(n, 0, "GE"),
    bitVecIntComp(r, 0, "EQ"),
    Equals(x, m),
    Equals(y, n),
    And(bitVecIntComp(m, N, "LT"), bitVecIntComp(n, N, "LT"))
])

post_condition = Equals(one, one)

cond = bitVecIntComp(y, 0, "GT")      # condição de controlo do ciclo

trans = And([
    Equals(prime(x), BVLShl(x, one)),
    Ite(Equals(BVAnd(y, one), one),
        And([
            Equals(prime(y), BVLShr(BVSub(y, one), one)),
            Equals(prime(r), BVAdd(r, x))
        ]),
        Equals(prime(y), BVLShr(y, one)))
    )
])

W = SAU(variables, pre_condition, post_condition, cond, trans)

from math import log2, ceil

bound = ceil(log2(N))

#Run
W.unfold(bound)
```

sucesso na tentativa 14

Misc

```
In [ ]:
def test():
    solver = Solver()
    a = Symbol('a', BVType(16))
    b = Symbol('b', BVType(16))
    solver.add_assertion(GE(BVToNatural(BVAdd(a, b)), Int(56)))
    solver.add_assertion(Equals(BVAnd(a, BVOne(word_len)), BVSub(SBV(1, word_len), SBV(0, word_len))))
    solver.add_assertion(Not(Equals(a, SBV(0, word_len))))
    # assertion 1: a = b
    res = solver.solve() # SAT (res == True)
    if res:
        print(solver.get_model())
        print('value a: {}'.format(solver.get_value(a)))
        print('value b: {}'.format(solver.get_value(b)))
test()
```

```
In [ ]:
def test2():
    solver = Solver()
    a = Symbol('a', BVType(16))
    b = Symbol('b', BVType(16))
    solver.add_assertion(Equals(BVSub(a, SBV(-1, word_len)), SBV(-2243, word_len)))
    # assertion 1: a = b
    res = solver.solve() # SAT (res == True)
    if res:
        print(solver.get_model())
        print('value a: {}'.format(solver.get_value(a)))
        print('value b: {}'.format(solver.get_value(b)))
test2()
```

```
a := 63292_16
value a: 63292_16
value b: 0_16
```

```
In [ ]:
def test3():
    solver = Solver()
    x = Symbol('x', INT)
    y = Symbol('y', INT)
    z = Symbol('z', INT)
    solver.add_assertion(Equals(x + y - z, Int(4353452)))
    solver.add_assertion(NotEquals(x, Int(0)))
    solver.add_assertion(NotEquals(y, Int(0)))
    solver.add_assertion(NotEquals(z, Int(0)))
    # assertion 1: a = b
    res = solver.solve() # SAT (res == True)
    if res:
        print(solver.get_model())
test3()
```

```
z := 1
y := -1
x := 4353454
```

```
In [ ]: def test4():
    solver = Solver()
    solver.reset_assertions()
    x = Symbol('x', INT)
    y = Symbol('y', INT)
    z = Symbol('z', INT)
    solver.add_assertion(Equals(Plus(Plus(x, y), z), Int(123392753)))
    solver.add_assertion(NotEquals(x, Int(0)))
    solver.add_assertion(NotEquals(y, Int(0)))
    solver.add_assertion(NotEquals(z, Int(0)))
    solver.add_assertion(Not(Equals(x, y)))
    solver.add_assertion(Not(Equals(y, z)))
    solver.add_assertion(Not(Equals(x, z)))
    # assertion 1: a = b
    res = solver.solve() # SAT (res == True)
    if res:
        print(solver.get_model())
test4()
```

```
z := -1
y := -2
x := 123392756
```

```
In [ ]: def test5():
    solver = Solver()
    a = Symbol('a', BVType(16))
    solver.add_assertion(NotEquals(a, BVZero(word_len)))
    solver.add_assertion(NotEquals(a, BV(65535, word_len)))
    prop = ForAll([a], Implies(NotEquals(a, BVZero(word_len)), NotEquals(BVZero(word_len), a)))
    solver.add_assertion(prop)
    res = solver.solve() # SAT (res == True)
    if res:
        print(solver.get_model())
        print('value a: {}'.format(solver.get_value(a)))
    else:
        print("Unsat!")
test5()
```

```
a := 1_16
value a: 1_16
```

```
In [ ]: format(45, 'b')
```

```
Out[ ]: '101101'
```

```
In [ ]: n = 59824
from math import log2
log2(n)
```

```
Out[ ]: 15.868436755858168
```

```
In [ ]: len(format(n, 'b'))
```

```
Out[ ]: 16
```