Pergunta1

January 5, 2022

1 Lógica Computacional 21/22

1.1 Trabalho Prático 4

ID: Pergunta 1 - Grupo 1

Equipa:

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

1.2 Preparação do código

1.2.1 Dependências

[ ]:
```
%pip install pysmt
```

```
Requirement already satisfied: pysmt in c:\programdata\anaconda3\lib\site-
packages (0.9.0)
Requirement already satisfied: six in c:\programdata\anaconda3\lib\site-packages
(from pysmt) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

1.3 Enunciado do problema

1.3.1 Código 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,

1. Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.
2. Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.
3. Construa a definição iterativa do "single assignment unfolding" usando um parâmetro limite

$$N$$

e aumentando a pré-condição com a condição

$$(n < N) \land (m < N)$$

O número de iterações vai ser controlado por este parâmetro

$$N$$

1.4 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:

1.4.1 Variáveis e estado inicial do FOTS

[ ]:
```
from pysmt.shortcuts import *
from pysmt.typing import *

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.

[ ]:
```
def bitVecIntComp(vec, i, bin_op):
    intVec = BV(i, word_len)
    if i == 0:
        intVec = BVZero(word_len)
    elif i == 1:
        intVec = BVOne(word_len)
    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 bitVecIntComp!
↪")
```

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.

[ ]:
```
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)
```

1.4.2 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:

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

e

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

$pc = 1$ 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:

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

e

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

$pc = 2$ 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.

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

$pc = 3$ Este caso correspondo à transição do estado final para ele próprio:

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

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

[ ]:
```
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'],  BVAdd(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)
```

1.4.3 Geração de traço de execução

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

[ ]:
```
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  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  0
state 1:
x  =  9377_16
y  =  0_16
```

```
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 2:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 3:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 4:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 5:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 6:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 7:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 8:
x  =  9377_16
```

```
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 9:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 10:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 11:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 12:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 13:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 14:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 15:
```

```
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 16:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 17:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 18:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
state 19:
x  =  9377_16
y  =  0_16
r  =  0_16
m  =  9377_16
n  =  0_16
pc  =  3
```

1.4.4 Prova de terminação do programa

Utilize-se k-lookahead para provar a terminação do programa acima, que é uma propriedade de segurança da forma $G\ \phi$.

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 estritamente positivo (ou nulo): $G\ (V(s) \geq 0)$
2. O variante é estritamente decrescente, ou atinge o valor 0, ou seja,

- $G\ (\forall s'\ .\ trans(s,\ s') \rightarrow (V(s') < V(s) \vee V(s') = 0))$

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 2 condição pode ser alargada, permitindo-se que o variante decresça de 3 em 3 transições.

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:

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

Comecemos por provar que V é sempre positivo:

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

def kinduction_always(declare, init, trans, inv, k):
    trace = [declare(i) for i in range(k + 1)]
    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]))

    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
```

[ ]:
```
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 0x0000028C16A69670>
```

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

    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
```

[ ]:
```
def term(state):
    return Implies(Equals(variante(state), Int(0)), Equals(state['pc'], Int(3)))
```

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

Verifica-se

```
[ ]: 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(-1, 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()
```

```
[ ]: 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
```

```
[ ]: 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
```

[ ]:
```
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
```

[ ]:
```
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(BVToNatural(a), Int(0))))
    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
```