

TP4 - Grupo 4

Pedro Paulo Costa Pereira - A88062

Tiago André Oliveira Leite - A91693

```
In [1]: from z3 import *
```

Problema - Correção de Programas

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
```

Variáveis Globais

- A variável Bits vai representar o tamanho de cada bitvector.
- A variável Limite vai definir qual o maior número que um bitvector pode conter por forma a evitar overflow.

```
In [2]: Bits = 16
        Limite = 8190
```

Função que declara as variáveis de cada estado

As variáveis são guardadas guardadas num dicionário *s*. Sendo que representam:

- $s[m'] \rightarrow$ um bitvector com o valor inicial de *x*;
- $s[n'] \rightarrow$ um bitvector com o valor inicial de *y*;
- $s[x'] \rightarrow$ um bitvector com o valor de *x*;
- $s[y'] \rightarrow$ um bitvector com o valor de *y*;
- $s[r'] \rightarrow$ um bitvector com o valor de *r*;
- $s[pc'] \rightarrow$ um bitvector com o valor do program counter;

```
In [3]: def declare(i):
        s = {}
        s['m'] = BitVec('m'+str(i),Bits)
        s['n'] = BitVec('n'+str(i), Bits)
        s['x'] = BitVec('x'+str(i), Bits)
        s['y'] = BitVec('y'+str(i), Bits)
        s['r'] = BitVec('r'+str(i), Bits)
        s['pc'] = BitVec('pc'+str(i), Bits)

        return s
```

Função que adiciona as restrições do estado inicial

As restrições no estado inicial são:

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

```
In [4]: def init(s):
        return And(s['m'] >= 0, s['m']<Limite, s['n'] >= 0, s['n']<Limite, s['r'] == 0, s['x'] == s['m'], s['y'] == s
```

Função que adiciona as restrições associadas a cada transição

As restrições associadas a cada transição são:

- $t_0 \rightarrow t_1 \equiv m = m' \wedge n = n' \wedge x = x' \wedge y = y' \wedge y > 0 \wedge pc = 0 \wedge pc' = 1 \wedge r = r'$
- $t_0 \rightarrow t_3 \equiv m = m' \wedge n = n' \wedge x = x' \wedge y = y' \wedge y \leq 0 \wedge pc = 0 \wedge pc' = 3 \wedge r' = r + x$
- $t_1 \rightarrow t_2 \equiv m = m' \wedge n = n' \wedge x = x' \wedge y' = y - 1 \wedge y \& 1 = 1 \wedge pc = 1 \wedge pc' = 2 \wedge r' = r + x'$

$$m = m' \wedge n = n' \wedge x = x' \wedge y = y' \wedge y \& 1 = 0 \wedge pc = 1 \wedge pc' = 2 \wedge r = r'$$
- $t_2 \rightarrow t_0 \equiv m = m' \wedge n = n' \wedge x' = x << 1 \wedge y' = y >> 1 \wedge pc = 2 \wedge pc' = 0 \wedge r = r'$
- $t_3 \rightarrow t_3 \equiv m = m' \wedge n = n' \wedge x = x' \wedge y = y' \wedge pc = 3 \wedge pc' = 3 \wedge r = r'$

Seja *trans* = [*X*₀, ..., *X*_i], o conjunto de todas as transições definidas em cima.

O resultado final é:

- $\bigvee_{i \in trans} X_i$

```
In [5]: def trans(s,p):
        t01 = And(s['m']==p['m'], s['n']==p['n'], s['x']==p['x'], s['y']==p['y'],
                  s['y']>0,s['pc']==0, p['pc']==1, s['r'] == p['r'])
        t03 = And(s['m']==p['m'], s['n']==p['n'], s['x']==p['x'], s['y']==p['y'],
                  s['y']<=0,s['pc']==0, p['pc']==3, s['r'] == p['r'])

        t12a = And(s['m']==p['m'], s['n']==p['n'], s['x']==p['x'], p['y']==s['y']-1,
                  s['y'] & 1 == 1,s['pc']==1, p['pc']==2, p['r'] == s['r']+s['x'])

        t12b = And(s['m']==p['m'], s['n']==p['n'], s['x']==p['x'], s['y']==p['y'],
                  s['y'] & 1 == 0,s['pc']==1, p['pc']==2, s['r'] == p['r'])

        t12 = Or(t12a, t12b)

        t20 = And(s['m']==p['m'], s['n']==p['n'], p['x']==s['x']<<1, p['y']==s['y']>>1,
                  s['pc']==2, p['pc']==0, s['r'] == p['r'])

        t33 = And(s['m']==p['m'], s['n']==p['n'], s['x']==p['x'], s['y']==p['y'],
                  s['pc']==3, p['pc']==3, s['r'] == p['r'])

        return Or(t01, t03, t12, t20, t33)
```

Função que gera um traço de execução do sistema com k estados

A função vai simular, através do solver Z3, uma execução do sistema com **k** estados, [*E*₀, .. *E*_{*k*-1}], e **k-1** transições, com as restrições codificadas anteriormente. Sendo *E*₀ o estado inicial.

Caso o sistema seja satisfazível, vão ser imprimidas, estado a estado, os valores das variáveis do sistema.

```
In [6]: def gera_traco(declare,init,trans, k):
        s = Solver()
        traco = {}
        for i in range(k):
            traco[i] = declare(i)
        s.add(init(traco[0]))
        test = []
        for i in range(k-1):
            s.add(trans(traco[i],traco[i+1]))
            status = s.check()
            if status == sat:
                m = s.model()
                for i in range(k):
                    for v in traco[i]:
                        print(v,"=",m[traco[i][v]])
                        print('')
            elif status == unsat:
                print("Não ha execuções possiveis")
            else:
                print("Resultado impossivel de obter!")
```

```
In [7]: gera_traco(declare,init,trans, 10)

m = 3954
n = 5171
x = 3954
y = 5171
r = 0
pc = 0

m = 3954
n = 5171
x = 3954
y = 5171
r = 0
pc = 1

m = 3954
n = 5171
x = 7908
y = 5170
r = 0
pc = 2

m = 3954
n = 5171
x = 7908
y = 2585
r = 3954
pc = 0

m = 3954
n = 5171
x = 7908
y = 2584
r = 11862
pc = 2

m = 3954
n = 5171
x = 15816
y = 1292
r = 11862
pc = 0

m = 3954
n = 5171
x = 15816
y = 1292
r = 11862
pc = 1

m = 3954
n = 5171
x = 31832
y = 646
r = 11862
pc = 0

m = 3954
n = 5171
x = 15816
y = 1292
r = 11862
pc = 2
```

1. Provar por indução a terminação deste programa

Para provar que o programa termina temos que encontrar um variante, que respeite as seguintes propriedades:

- O variante nunca é negativo, ou seja, $V(s) \geq 0$
- O variante decresce sempre (estritamente) até atinge o valor 0, ou seja, $\forall s'. trans(s, s') \rightarrow V(s') < V(s) \vee V(s') = 0$
- Quando o variante é 0 verifica-se necessariamente $pc = 4$, ou seja, $GV(s) = 0 \rightarrow s[pc] = 4$.

Função que prova uma propriedade por k indução

```
In [8]: def kinduction_always(declare,init,trans,prop,k):
        s = Solver()
        state = {}
        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(prop(state[i])) for i in range(k)]))

        status = s.check()
        assert(status != unknown)
        if (status == sat):
            print("Não é verdade nos estados iniciais")
            m = s.model()
            for i in range(k):
                print(i)
                for v in state[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(prop(state[i]))
                s.add(trans(state[i],state[i+1]))
            s.add(Not(prop(state[k])))
            status = s.check()
            if (status == sat):
                print("0 passo indutivo não é verdade nos estados")
                m = s.model()
                for i in range(k):
                    print(i)
                    for v in state[i]:
                        print(v, '=', m[state[i][v]])

            return
        print("0 invariante é valido!")
```

Variante

```
In [9]: def variante(state):
        return 3*state['y']-state['pc']+3
```

Propriedades do variante

Não negativo

```
In [10]: def naonegativo(state):
        return variante(state) >=0
```

```
In [11]: kinduction_always(declare,init,trans,naonegativo,20)

0 invariante é valido!
```

Decrescente

```
In [12]: def decrescente(state):
        proximo = declare(-1)
        decresce = variante(proximo) < variante(state)
        zero = Or(proximo['y']==0,variante(proximo) == 0)
        formula = Implies(trans(state,proximo),Or(zero,decresce))
        return (ForAll(list(proximo.values()),formula))
```

```
In [13]: kinduction_always(declare,init,trans,decrescente,20)

0 invariante é valido!
```

Util

```
In [14]: def utilidade(state):
        return (Implies(variante(state) == 0, state['pc'] == 3))
```

```
In [15]: kinduction_always(declare,init,trans,utilidade,20)

0 invariante é valido!
```

2. Correção total

a. Forma recursiva do programa codificado em LPA (linguagem de programas anotados).

$P \equiv$ assume ϕ ; W ; assert φ

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

$\varphi \equiv r = m \times n$

$W \equiv$ assume *b*; *S*; $W \parallel$ assume $\neg b$; skip

$b \equiv y > 0$

$S \equiv$ assume($y \& 1 = 1$); $y \leftarrow y - 1$; $r \leftarrow r + x$; $C \parallel$ assume $\neg(y \& 1 = 1)$; C

$C \equiv x \leftarrow x << 1$; $y \leftarrow y >> 1$

b. Invariante mais fraco que assegura a correção do programa.

Para provar a correção do programa temos que encontrar um invariante de ciclo, θ , que respeite as seguintes propriedades:

- Inicialização, ou seja, $\phi \rightarrow \theta$;
- Preservação, ou seja, $W \wedge \theta \rightarrow \theta$;
- Utilidade, ou seja, $\neg b \wedge \theta \rightarrow \varphi$;

Um cadidato a invariante de ciclo pode ser:

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

Função que prova as propriedades

```
In [16]: def prove(f):
        s = Solver()
        s.add(Not(f))
        c = s.check()
        if c == unsat:
            print("A propriedade é válida!")
        elif c == sat:
            print("A propriedade é inválida!")
            print(s.model())
        else:
            print("Resultado desconhecido")
```

Variáveis

```
In [17]: m, n, r, x, y = BitVecs('m n r x y',Bits)
```

Invariante

```
In [18]: invariante = And(x*y+r==m*n,y>=0, n>y,m>=0,n>=0)
```

Inicialização

```
In [19]: pre = And(m>=0,n>=0,r==0,x==0,y==n)
        prove(Implies(pre,invariante))

A propriedade é válida!
```

Preservação

```
In [20]: vcthen = Implies(y & 1 == 1, substitute (substitute(substitute(invariante,(y,y>>1)),(x,x<<1)),(r,r+x)
vcelse = Implies(y & 1 == 0, substitute (substitute(invariante,(y,y>>1)),(x,x<<1)))
cycle = And(vcthen,vcelse)
prove(Implies(And(invariante,cycle),invariante))

A propriedade é válida!
```

Utilidade

```
In [21]: not_b = y==0
        pos = r==m*n
        prove(Implies(And(not_b,invariante),pos))

A propriedade é válida!
```

c. Unfold com parametro limite N para provar correção do progma.

Para provar a correção por unfold com um parametro limite **N**, temos que "esticar" (**unfold**) a execução do ciclo numa aproximação formada por uma sequência finita, numero finito, de execuções do "corpo do ciclo".

Orta, sendo **N** o parametro limite e uma vez que em cada execução do ciclo existe uma sequencia de no máximo 3 estados, sabemos então que o ciclo vai executar um número finito de vezes. Portanto, podemos definir um número maximo de $3 \times N + 1$ estados.

Assim sendo, para provar o invariante, vamos gerar um traço de execução do programa com $3 \times N + 1$ estados, [*S*₀ .. *S*_k],onde, sendo serão acrescentadas a seguintes restrições:

- $S_0[m'] < N \wedge S_0[n'] < N$
- $\forall i \in [0, N] i \% 3 = 0 \implies S_i[y'] > 0 \vee (S_i[y'] = 0 \wedge \theta(S_i))$
- $\neg \theta(S_k)$

Por fim utilizamos o solver do z3 para realizar o traço de execução e caso o **status** seja **unsat** isso significa que o programa está correto.

Pré-condição

```
In [22]: def pre(s,N):
        return And(s['m'] >= 0,s['m']<N, s['n'] >= 0,s['n']<N, s['r'] == 0, s['x'] == s['m'], s['y'] == s['n'], s['
```

Condição do ciclo

```
In [23]: def b(s):
        return Or(s['y'] > 0,And(s['y']==0,s['r']==s['m']*s['n']))
```

Pós-condição

```
In [24]: def pos(s):
        return And(s['r'] == s['m']*s['n'])
```

Função que prova a correção do programa por unfold

```
In [25]: def unfold(declare,pre,trans,b,pos,N):
        k = N*3 + 1
        s = Solver()
        traco = {}
        for i in range(k):
            traco[i] = declare(i)
        s.add(pre(traco[0],N))
        for i in range(k-1):
            s.add(trans(traco[i],traco[i+1]))
            if i%3 == 0:
                s.add(b(traco[i]))
            s.add(Not(pos(traco[k-1])))
        status = s.check()
        if status == sat:
            m = s.model()
            for i in range(k):
                print(i)
                for v in traco[i]:
                    print(v, "=", m[traco[i][v]])

            return
        print("O programa está correto!")
```

```
In [26]: unfold(declare,pre,trans,b,pos,20)

O programa está correto!
```

Função que prova a correção do programa por unfold usando a função prove

```
In [27]: def unfold_prove(declare,pre,trans,b,pos,N):
        k = N*3 +1
        traco = {}
        for i in range(k):
            traco[i] = declare(i)
        pre_condition = pre(traco[0],N)
        cycle = []
        for i in range(k-1):
            cycle.append(trans(traco[i],traco[i+1]))
            if i%3 == 0:
                cycle.append(b(traco[i]))
        cycle = And(cycle)
        pos_condition = ((pos(traco[k-1])))
        formula = Implies(And(pre_condition,cycle),pos_condition)
        prove(formula)
```

```
In [28]: unfold_prove(declare,pre,trans,b,pos,20)

A propriedade é válida!
```