Lógica Computacional: 21/22

Trabalho 4

Grupo 7

- David José de Sousa Machado (A91665)
- Ivo Miguel Gomes Lima (A90214)

Inicialização

Para a resolução destes exercícios utilizamos as bibliotecas:

- Python Z3Py
- Satisfiability Modulo Theory

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

```
from z3 import *
import pysmt.shortcuts as ps
import pysmt.typing as pt
```

Contextualização do Problema

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

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

O número de iterações vai ser controlado por este parâmetro N

0. Análise

Antes de começámos a resolução codificámos o programa e fizemos print dos valores das variáveis para termos nos ajudar a descobrir o **variante** e o **invariante** do mesmo.

```
def problema(x, y):
    r = 0
    print("x y r")
    print(x, y, r, "entrada")

    # y >= 0 and x*y+r == n*m
    while y > 0:
        if y & 1 == 1:
            y, r = y-1, r+x
            print(x, y, r, "if")
        x, y = x<<1, y>>1
        print(x, y, r, "while")

    print(x, y, r, "final")
    return r

problema(2, 3)
```

```
x y r
2 3 0 entrada
2 2 2 if
4 1 2 while
4 0 6 if
8 0 6 while
8 0 6 final
6
```

1. Prove por indução a terminação deste programa

Para usarmos indução e provarmos que o programa apresentado termina vamos construir um *FOTS* que o modela. Temos então as variáveis `m`, `n`, `x`, `y`, `r` que irão fazer parte do *FOTS*.

Podemos ainda considerar uma variável `pc` que nos indicará em que instrução nos encontramos.

Define-se então a função que declara as variáveis:

```
def declare(i):
    return {
        "pc": Int("pc"+str(i)),
        "m": BitVec("m"+str(i), 16),
        "n": BitVec("n"+str(i), 16),
```

```
        "x": BitVec("x"+str(i), 16),
        "y": BitVec("y"+str(i), 16),
        "r": BitVec("r"+str(i), 16)
    }
```

Para definirmos o predicado `init` que determina o estado inicial do *FOTS* basta olharmos para a pré-condição do programa que diz `assume m >= 0 and n >= 0 and r == 0 and x == m and y == n`.

Portanto o `init` será

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

```
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"])
```

Define-se agora a função de transição.

Se a variável `pc` tiver o valor 0, chegamos à condição do ciclo, onde podem correr 3 situações:

- Estamos dentro do ciclo e a condição do if é verificada fazendo-nos entrar no corpo do mesmo.
$$pc == 0 \land\ pc' == 1 \land\ m' == m \land\ n' == n \land\ x' == x \land\ y' == y \land\ r' == r \land\ y > 0 \land\ y \,\&\, 1 == 1$$
- Estamos dentro do ciclo mas a condição if não é verificada levando-nos a não entrar no corpo do ciclo.
$$pc == 0 \land\ pc' == 2 \land\ m' == m \land\ n' == n \land\ x' == x \land\ y' == y \land\ r' == r \land\ y > 0 \land\ y \,\&\, 1 \neq 1$$
- Ir para o fim do programa caso a condição de ciclo seja falsa.
$$pc == 0 \land\ pc' == 3 \land\ m' == m \land\ n' == n \land\ x' == x \land\ y' == y \land\ r' == r \land\ y \leq 0$$

Se a variável `pc` tiver o valor 1, estamos dentro do ciclo, onde ocorre a alteração das variáveis `y` e `r`.

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

Se a variável `pc` tiver o valor 2, temos apenas de executar as últimas instruções do ciclo e colocar o valor de `pc` a 0, para testarmos novamente a condição de ciclo.

$$pc == 2 \land\ pc' == 0 \land\ m' == m \land\ n' == n \land\ x' \ll 1 \land\ y' == y \gg 1 \land\ r' == r$$

Se a variável `pc` tiver o valor 3, temos de adicionar o lacete final, em que o estado final transita para ele próprio.

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

```
def trans(curr,prox):
    preserve = lambda x : prox[x] == curr[x]
    preserveAll = And(preserve("m"), preserve("n"), preserve("x"), preserve("y"), preserve("r"))

    t01 = And(curr["pc"] == 0, prox["pc"] == 1, preserveAll, curr["y"] > 0, curr["y"] & 1 == 1)
    t02 = And(curr["pc"] == 0, prox["pc"] == 2, preserveAll, curr["y"] > 0, curr["y"] & 1 != 1)
    t03 = And(curr["pc"] == 0, prox["pc"] == 3, preserveAll, curr["y"] <= 0)
    t12 = And(curr["pc"] == 1, prox["pc"] == 2, preserve("m"), preserve("n"), preserve("x"), prox["y"] == curr["y"] - 1, prox["r"] == curr["r"] + curr["x"])
    t20 = And(curr["pc"] == 2, prox["pc"] == 0, preserve("m"), preserve("n"), prox["x"] == curr["x"] << 1, prox["y"] == curr["y"] >> 1, preserve("r"))
    t33 = And(curr["pc"] == 3, prox["pc"] == 3, preserveAll)

    return Or(t01, t02, t03, t12, t20, t33)
```

```
# transição para quando a condição do if está separada do seu corpo
def transAlt(curr,prox):
    t01 = And(curr["pc"] == 0, prox["pc"] == 1, curr["y"] > 0, prox["x"] == curr["x"], prox["y"] == curr["y"], prox["r"] == curr["r"])
    t04 = And(curr["pc"] == 0, prox["pc"] == 4, curr["y"] <= 0, prox["x"] == curr["x"], prox["y"] == curr["y"], prox["r"] == curr["r"])
    t12 = And(curr["pc"] == 1, prox["pc"] == 2, curr["y"] & BitVecVal(1,16) == BitVecVal(1,16), prox["x"] == curr["x"], prox["y"] == curr["y"], prox["r"] == curr["r"])
    t13 = And(curr["pc"] == 1, prox["pc"] == 3, curr["y"] & BitVecVal(1,16) != BitVecVal(1,16), prox["x"] == curr["x"], prox["y"] == curr["y"], prox["r"] == curr["r"])
    t23 = And(curr["pc"] == 2, prox["pc"] == 3, prox["x"] == curr["x"], prox["y"] == curr["y"] - BitVecVal(1,16), prox["r"] == curr["r"] + curr["x"])
    t30 = And(curr["pc"] == 3, prox["pc"] == 0, prox["x"] == curr["x"] << BitVecVal(1,16), prox["y"] == curr["y"] >> BitVecVal(1,16), prox["r"] == curr["r"])
    t44 = And(curr["pc"] == 4, prox["pc"] == 4, prox["x"] == curr["x"], prox["y"] == curr["y"], prox["r"] == curr["r"])

    return Or(t01, t04, t12, t13, t23, t30, t44)
```

Antes de utilizamos a indução para demonstrar que o programa termina. Devemos começar por encontrar um **variante** V que satisfaça as seguintes condições:

- Ser sempre positivo, ou seja, $V_s \geq 0$
- O variante atinge o valor de 0 ou é estritamente descrescente, isto é , $\forall_s' \cdot trans(s,s') \rightarrow (V_{s'} < V_s \vee V_{s'} = 0)$
- Em 0 o variante verifica ϕ, temos portanto que $V_s = 0 \rightarrow \phi_s$ sendo o ϕ a chegada à pós-condição.

Como vamos queremos uma propriedade *liveness* vamos utilizar a `lookahead`, acabamos por relaxar a 2º condição, permitindo que o **variante** diminua de 3 em 3 transições. Consideramos um `lookahead` de 3 pois é o valor que nos permite *saltar* o corpo do ciclo na última iteração.

Após a análise das condições podemos considerar que o **variante** V do ciclo será:

$$V_s = y_s - pc_s + 3$$

De seguida apresentamos as codificações do variante assim como as condições a serem verificadas.

```
def variante(state):
    return BV2Int(state["y"]) + 3 - state["pc"]
```

Condição de ser sempre positivo.

```
def nao_negativo(state):
    return (variante(state) >= 0)
```

Condição do variante ser estritamente descrescente ou atingir o valor de 0.

```
def decrescente(state):
    prox1 = declare(-1)
    prox2 = declare(-2)
    prox3 = declare(-3)

    decresce = variante(prox3) < variante(state)
    zero = variante(prox3) == 0

    formula = Implies(
        And(trans(state, prox1), trans(prox1, prox2), trans(prox2, prox3)),
        Or(zero, decresce)
    )
```

```
    return ForAll(list(prox1.values()) + list(prox2.values()) + list(prox3.values()), formula)
```

Quando o variante chega a 0 verificamos que terminamos o ciclo.

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

Depois da definição de todas estas condições podemos finalmente provar por indução a válidade das mesmas através do `kinduction_always`.

```
def dump_die(m, state, k):
    for i in range(k):
        print("i =", i)
        for v in state[i]:
            print(v, "=", m[state[i][v]])
        print()
    print()

def kinduction_always(declare,init,trans,inv,k):
    s = Solver()
    state = {i: declare(i) for i in range(k)}

    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)]))

    if s.check() == sat:
        m = s.model()
        print("Não é verdade nos estados iniciais")
        dump_die(m, state, k)
        return False

    s = Solver()
    state = {i: declare(i) for i in range(k+1)}

    for i in range(k):
        s.add(inv(state[i]))
        s.add(trans(state[i], state[i+1]))

    s.add(Not(inv(state[k])))

    if s.check() == sat:
        m = s.model()
        print("O passo indutivo não é verdade nos estados")
        dump_die(m, state, k)
        return False

    print("O invariante é válido")
    return True
```

```python
def exec_termina(nao_negativo, utilidade, decrescente):
    cond = False
    k = 0
    max = 50

    while cond == False and k < max:
        k += 1

        cond = kinduction_always(declare,init,trans,nao_negativo,k)
        cond = cond and kinduction_always(declare,init,trans,utilidade,k)
        cond = cond and kinduction_always(declare,init,trans,decrescente,k)

    return k

exec_termina(nao_negativo, utilidade, decrescente)
```

````
# 2. a.

Codifique usando a **LPA** (linguagem de programas anotadas) a forma recursiva 
deste programa.

$$
W \equiv \{\mathsf{assume}\ (y > 0);\ S;\ W\}\ \|\ \{\mathsf{assume}\ (y \le 0)
\}\\
S ≡ \{\mathsf{assume}\ (y\ \&\ 1 = 1);\ C;\ Z\}\ \|\ \{\mathsf{assume}\ (y\ \&\ 
1 \ne 1);\ Z\}\\
C ≡ \{\mathit{y} \gets \mathit{y - 1};\ \mathit{r} \gets \mathit{r + x}\}\\
Z ≡ \{\mathit{y} \gets \mathit{y \gg 1};\ \mathit{x} \gets \mathit{x \ll 1}\}
$$

```python
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
assert inv;
# Ciclo
havoc x; havoc y; havoc r;

#Corpo
((assume (y > 0) and inv;
    ((assume (y & 1 == 1);
        y = y-1;
        r = r+x;
    ) || (
        assume (not (y & 1 == 1));
        skip;
    ))
    x = x<<1;
    y = y>>1;
    assert inv;
    assume False;
) || (
    assume (not (y > 0)) and inv;
````


2. a.

Codifique usando a **LPA** (linguagem de programas anotadas) a forma recursiva deste programa.

$$
W \equiv \{\mathsf{assume}\ (y > 0);\ S;\ W\}\ \|\ \{\mathsf{assume}\ (y \le 0)\}\\
S \equiv \{\mathsf{assume}\ (y\ \&\ 1 = 1);\ C;\ Z\}\ \|\ \{\mathsf{assume}\ (y\ \&\ 1 \ne 1);\ Z\}\\
C \equiv \{\mathit{y} \gets \mathit{y - 1};\ \mathit{r} \gets \mathit{r + x}\}\\
Z \equiv \{\mathit{y} \gets \mathit{y \gg 1};\ \mathit{x} \gets \mathit{x \ll 1}\}
$$

```python
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
assert inv;
# Ciclo
havoc x; havoc y; havoc r;

# Corpo
((assume (y > 0) and inv;
    ((assume (y & 1 == 1);
        y = y-1;
        r = r+x;
    ) || (
        assume (not (y & 1 == 1));
        skip;
    ))
    x = x<<1;
    y = y>>1;
    assert inv;
    assume False;
) || (
```
```

```
));

assert·r·==·m·*·n;
#·End·Ciclo/Corpo
```

```
    assume (not (y > 0)) and inv;
));

assert r == m * n;
# End Ciclo/Corpo
```

2. b.

Temos agora de definir um invariante que assegure a correção, para o efeito definimo-lo da seguinte forma:

$$m * n == x * y + r \ \land \ y \geq 0$$

Após este passo vamos agora utilizar as regras da metodologia WPC com o intuito de gerar a condição de verificação:

$$\begin{array}{l}
\textsf{inv} = y >= 0 \land x * y + r == m * n \\
\textsf{pre} = m >= 0 \land n >= 0 \land r == 0 \land x == m \land y == n \\
\textsf{pos} = r == m * n \\
\\
\textsf{pre} \to (\textsf{inv} \land [\textsf{Ciclo}]) \\
\equiv \quad \text{(havoc)} \\
\textsf{pre} \to (\textsf{inv} \land \forall x. \forall y. \forall r. \, [\textsf{Corpo}]) \\
\equiv \quad (\text{porque } x \text{ e } y \text{ estão quantificadas}) \\
\textsf{pre} \to \textsf{inv} \land \forall x. \forall y. \forall r. \, [\textsf{Corpo}] \\
\equiv \quad (\text{porque assume } \mathit{False} \text{ dá origem a } \mathit{False} \to \ldots = \mathit{True}) \\
\textsf{pre} \to \textsf{inv} \ \land \ (\forall x. \forall y. \forall r. \ (\\
\quad y > 0 \land \textsf{inv} \to (\\
\quad\quad (y \,\&\, 1 = 1 \to \textsf{inv}[(y \gg 1)/y][(x \ll 1)/x][(r+x)/r][(y-1)/y]) \land \\
\quad\quad (\neg(y \,\&\, 1 = 1) \to \textsf{inv}[(y \gg 1)/y][(x \ll 1)/x]) \\
\quad) \land (\neg(y > 0) \land \textsf{inv} \to \textsf{pos}) \\
))
\end{array}$$

Desta forma podemos traduzir o programa na seguinte linguagem de fluxos:

```
  [assume m >= 0 and n >= 0 and r == 0 and x == m and y == n; assert inv;
  havoc x; havoc y; havoc r;
  (assume y>0 and inv; (assume y & 1==1; y=y-1; r= r+x;
  || assume not (y & 1==1);skip;)x=x<<1; y=y>>1;
  assert inv; assume False;
  || assume not(y>0) and inv;)
  assert r == m * n]

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  [assert inv; havoc x; havoc y; havoc r;
  (assume y>0 and inv;(assume y & 1==1; y=y-1; r= r+x;
```

```
  || assume not (y & 1==1);) x=x<<1; y=y>>1;
  assert inv; assume False; assert r == m * n;
  || assume not(y>0) and inv; assert r == m * n;)]

= m >= 0 and n >= 0 and r == 0 and x == m and y == n => (inv and
  [havoc x; havoc y; havoc r;
  (assume y>0 and inv; (assume y & 1==1; y=y-1; r= r+x;
  || assume not (y & 1==1);) x=x<<1; y=y>>1;
  assert inv; assume False; assert r == m * n;
  || assume not(y>0) and inv;assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r
  [(assume y>0 and inv; (assume y & 1==1; y=y-1; r= r+x;
  || assume not (y & 1==1);) x=x<<1; y=y>>1;
  assert inv; assume False; assert r == m * n;
  || assume not(y>0) and inv;assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r
  (y>0 and inv => [(assume y & 1==1; y=y-1; r= r+x;
  || assume not (y & 1==1);)x=x<<1; y=y>>1;
  assert inv; assume False; assert r == m * n;
  || assume not(y>0) and inv; assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r
  (y>0 and inv => [(assume y & 1==1; y=y-1; r= r+x; x=x<<1; y=y>>1;
  assert inv; assume False; assert r == m * n;
  || assume not (y & 1==1); x=x<<1; y=y>>1; assert inv;
  assume False; assert r == m * n;)
  || assume not(y>0) and inv; assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r
  (y>0 & inv => [((assume y and 1==1; (assert inv; assume False;
  assert r == m * n;)[y>>1/y] [x<<1/x][r+x/r][y-1/y])
  || (assume not (y & 1==1);
  (assert inv;assume False; assert r == m * n;)[y>>1/y][x<<1/x])
  || assume not(y>0) and inv; assert r == m * n;)])

= m>=0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r
  (y>0 and inv => ((y & 1==1 => (inv and (False => r == m * n;))
  [y>>1/y][x<<1/x][r+x/r][y-1/y]) and [(assume not (y & 1==1);
  (assert inv; assume False; assert r == m * n;)[y>>1/y] [x<<1/x])
```

```
  and assume not(y>0) and inv; assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>
  (inv and forall x forall y forall r (y>0 and inv =>
  ((y & 1==1 => (inv and (False => r == m * n))
  [y>>1/y][x<<1/x][r+x/r][y-1/y]) and (not (y & 1==1) =>
  (inv and (False => r == m * n))[y>>1/y][x<<1/x]) and
  [assume not(y>0) and inv; assert r == m * n;)])

= m>=0 and n>=0 and r==0 and x==m and y==n =>
  inv and forall x forall y forall r
  (y>0 and inv => (y & 1==1 =>
  (inv and (False => r == m * n)) [y>>1/y][x<<1/x][r+x/r][y-1/y]) and
  (not (y & 1==1) => (inv and (False => r == m * n))[y>>1/y][x<<1/x]) and
  not(y>0) and inv => r == m * n)

= m>=0 and n>=0 and r==0 and x==m and y==n =>
  inv and forall x forall y forall r (y>0 and inv =>
  (y & 1==1 => (inv[y>>1/y][x<<1/x][r+x/r][y-1/y]) and
  (not (y & 1==1) =>(inv[y>>1/y][x<<1/x]) and not(y>0) and inv => r == m * n)
```

De seguida implementamos esta fórmula no $z3$ e tentamos provar a sua veracidade:

```
def prove(f):
    s = Solver()
    s.add(Not(f))
    r = s.check()
    if r == unsat:
        print("Proved")
    else:
        print("Failed to prove")
        m = s.model()
        for v in m:
            print(v,'=', m[v])
```

```
# Não termina para 16 bits
x, y, r, m, n = BitVecs('x y r m n', 8)
Zero = BitVecVal(0, 8)
One = BitVecVal(1, 8)

inv = And(y >= Zero, x*y+r == m*n)
pre = And(m >= Zero, n >= Zero, r == Zero, x == m, y == n)
pos = And(r == m * n)

fluxoIf = And(
    Implies(y&One == One, substitute(inv, [(y, y>>One), (x, x<<One), (r, r+x), (y, y-One)])),
    Implies(Not(y&One == One), substitute(inv, [(y, y>>One), (x, x<<One)]))
)
```

```
fluxoWhile = And(
    Implies(And(y > Zero, inv), fluxoIf),
    Implies(And(Not(y > Zero), inv), pos)
)

vc = Implies(pre, And(inv, ForAll([x, y, r], fluxoWhile)))

prove(vc)
```

```
Proved
```

E agora no PySMT:

```
def proveSMT(formula):
    print("Serialization of the formula:")
    print(formula)

    with ps.Solver(name="z3") as solver:
        solver.add_assertion(ps.Not(formula))
        if not solver.solve():
            print("Proved")
        else:
            print("Failed to prove")
```

```
bits = 8
zero = ps.BVZero(bits)
one  = ps.BVOne(bits)

# O ciclo
m = ps.Symbol("m", pt.BV8)
n = ps.Symbol("n", pt.BV8)
x = ps.Symbol("x", pt.BV8)
y = ps.Symbol("y", pt.BV8)
r = ps.Symbol("r", pt.BV8)

variables = [m, n, x, y, r]

inv = ps.And(y >= zero, ((x * y) + r).Equals(m * n))
pre = ps.And(m >= zero, n >= zero, r.Equals(zero), x.Equals(m), y.Equals(n))
pos = r.Equals(m * n)

subWhile = {
    y: ps.BVLShr(y, one),
    x: ps.BVLShl(x, one)
}
subIf = {
    y: ps.BVSub(y, one),
    r: ps.BVAdd(r, x)
}
fluxoIf = ps.And(
    ps.Implies(ps.BVAnd(y, one).Equals(one), inv.substitute(subWhile).substitute(subIf)),
    ps.Implies(ps.Not(ps.BVAnd(y, one).Equals(one)), inv.substitute(subWhile))
)
```

```
fluxoWhile = ps.And(
    ps.Implies(ps.And(y > zero, inv), fluxoIf),
    ps.Implies(ps.And(ps.Not(y > zero), inv), pos)
)

vc = ps.Implies(pre, ps.And(inv, ps.ForAll([x, y, r], fluxoWhile)))


proveSMT(vc)
```

```
Serialization of the formula:
(((0_8 u<= m) & (0_8 u<= n) & (r = 0_8) & (x = m) & (y = n)) -> (((0_8 u<= y) & ((... + ...) = (... * ...))) & (forall x, y, r . ((... -> ...) & (... -> ...)))))
Proved
```

2.c

Para a construção da definição iterativa do *single assignment unfolding* usamos um parâmetro limite N e aumentamos a pré-condição com a condição:

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

O número de iterações foi controlado pelo parâmetro N. Este N será o número máximo que é possível representar com b bits.
$N = 2^{b-1} - 1$.

Desenrolar uma vez

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
if (y0 > 0):
    if (y0 & 1 == 1):
        y1 = y0-1;
        r1 = r0+x0;
    else:
        y1 = y0;
        r1 = r0;
    x1, y2 = x0<<1, y1>>1;
    assert not (y2 > 0);
else:
    r1 = r0;
assert r1 == m0 * n0;
```

Desenrolar duas vezes

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n;
if (y0 > 0):
    if (y0 & 1 == 1):
        y1 = y0-1;
        r1 = r0+x0;
```

```
    else:
        y1 = y0;
        r1 = r0;
    x1, y2 = x0<<1, y1>>1;

    if (y2 > 0):
        if (y2 & 1 == 1):
            y3 = y2-1;
            r2 = r1+x1;
        else:
            y3 = y2;
            r2 = r1;
        x2, y4 = x1<<1, y3>>1;
        assert not (y4 > 0);
    else:
        x2 = x1;
        y4 = y2;
        r2 = r1;
else:
    x1 = x0;
    y4 = y0;
    r2 = r0;
assert r2 == m0 * n0;
```

Após estas tentativas notámos um padrão e decidimos escrever funções que nos vão dar o output.

```
def ifCode(lvl):
    return f'''
{tab*lvl}if (y{lvl*2} > 0):
{tab*lvl}    if (y{lvl*2} & 1 == 1):
{tab*lvl}        y{lvl*2+1} = y{lvl*2}-1;
{tab*lvl}        r{lvl+1} = r{lvl}+x{lvl};
{tab*lvl}    else:
{tab*lvl}        y{lvl*2+1} = y{lvl*2};
{tab*lvl}        r{lvl+1} = r{lvl};
{tab*lvl}    x{lvl+1} = x{lvl}<<1;
{tab*lvl}    y{lvl*2+2} = y{lvl*2+1}>>1;'''

def elseCode(lvl, N):
    return f'''
{tab*lvl}else:
{tab*lvl}    x{N} = x{lvl};
{tab*lvl}    y{N*2} = y{lvl*2};
{tab*lvl}    r{N} = r{lvl};'''


def unfoldCode(N, tab, lvl = 0):
    for i in range(N):
        print(ifCode(i))
```

```
    print(f"{tab*N}assert not (y{N*2} > 0);")

    for i in range(N-1, -1, -1):
        print(elseCode(i, N))

    return None


tab = ' '*4
unfoldCode(2, tab)
```

```
def fluxo(lvl):
    return f'''
assume (y{lvl*2} > 0);
((assume (y{lvl*2} & 1 == 1);
    y{lvl*2+1} = y{lvl*2}-1;
    r{lvl+1} = r{lvl}+x{lvl};
) || (assume not (y{lvl*2} & 1 == 1);
    y{lvl*2+1} = y{lvl*2};
    r{lvl+1} = r{lvl};
))
x{lvl+1} = x{lvl}<<1;
y{lvl*2+2} = y{lvl*2+1}>>1;'''

def unfoldFluxo(N):
    print("assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;")

    for lvl in range(N):
        print(fluxo(lvl))

    print(f"\nassert r{N} == m0 * n0 and not(y{N*2} > 0);")


unfoldFluxo(2)
```

```
ssume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;


((assume (y0 > 0);
    ((assume (y0 & 1 == 1);
        y1 = y0-1;
        r1 = r0+x0;
    ) || (assume not (y0 & 1 == 1);
        y1 = y0;
        r1 = r0;
    ))
    x1 = x0<<1;
    y2 = y1>>1;

    ((assume (y2 > 0);
```

```
        ((assume (y2 & 1 == 1);
            y3 = y2-1;
            r2 = r1+x1;
        ) || (assume not (y2 & 1 == 1);
            y3 = y2;
            r2 = r1;
        ))
        x2 = x1<<1;
        y4 = y3>>1;
    ) || (assume not(y2 > 0);
        x2 = x0;
        y4 = y0;
        r2 = r0;
    ))

) || (assume not(y0 > 0);
    x2 = x0;
    y4 = y0;
    r2 = r0;
))

assert r2 == m0 * n0 and not(y4 > 0);
```

```
def unfoldFluxoTotal(N):
    bits = N
    m0, n0 = BitVecs('m0 n0', bits)
    x_ = [BitVec(f'x{i}', bits) for i in range(N+1)]
    y_ = [BitVec(f'y{i}', bits) for i in range(N*2+2)]
    r_ = [BitVec(f'r{i}', bits) for i in range(N+1)]
    ciclo = True

    for i in range(N-1, -1, -1):
        fluxoIf = Or(
            And(y_[i*2] & 1 == 1, y_[i*2+1] == y_[i*2]-1, r_[i+1] == r_[i]+x_[i]),
            And(Not(y_[i*2] & 1 == 1), y_[i*2+1] == y_[i*2], r_[i+1] == r_[i])
        )

        ciclo = Or(
            And(y_[i*2] > 0, fluxoIf, x_[i+1] == x_[i]<<1, y_[i*2+2] == y_[i*2+1]>>1, ciclo),
            And(Not(y_[i*2] > 0), x_[N] == x_[i], y_[N*2] == y_[i*2], r_[N] == r_[i])
        )

    print(simplify(ciclo))

    pre = And(m0 >= 0, n0 >= 0, r_[0] == 0, x_[0] == m0, y_[0] == n0)
    pos = And(r_[N] == m0 * n0)

    vctt = Implies(
        And(pre, ciclo),
```

```
        And(pos, Not(y_[N*2] > 0))
    )

    prove(vctt)

unfoldFluxoTotal(8)
```

```
def fluxo2z3(x_, y_, r_, i):
    fluxoIf = Or(
        And(y_[i*2] & 1 == 1, y_[i*2+1] == y_[i*2]-1, r_[i+1] == r_[i]+x_[i]),
        And(Not(y_[i*2] & 1 == 1), y_[i*2+1] == y_[i*2], r_[i+1] == r_[i])
    )

    return And(y_[i*2] > 0, fluxoIf, x_[i+1] == x_[i]<<1, y_[i*2+2] == y_[i*2+1]>>1)

def unfoldFluxoThen(N):
    bits = N
    m0, n0 = BitVecs('m0 n0', bits)
    x_ = [BitVec(f'x{i}', bits) for i in range(N+1)]
    y_ = [BitVec(f'y{i}', bits) for i in range(N*2+2)]
    r_ = [BitVec(f'r{i}', bits) for i in range(N+1)]

    pre = And(m0 >= 0, n0 >= 0, r_[0] == 0, x_[0] == m0, y_[0] == n0)
    pos = And(r_[N] == m0 * n0)
    ciclo = And([fluxo2z3(x_, y_, r_, i) for i in range(N)])

    vctt = Implies(
        And(pre, ciclo),
        And(pos, Not(y_[N*2] > 0))
    )

    prove(vctt)

unfoldFluxoThen(16)
```

```
# PySMT example https://github.com/pysmt/pysmt
# https://pysmt.readthedocs.io/en/latest/_modules/pysmt/shortcuts.html
# https://pysmt.readthedocs.io/en/latest/_modules/pysmt/typing.html

def prime(v):
    return ps.Symbol("next(%s)" % v.symbol_name(), v.symbol_type())

def fresh(v):
    return ps.FreshSymbol(typename=v.symbol_type(),template=v.symbol_name()+"_%d")

class EPU(object):
    """deteção de erro"""

    def __init__(self, variables, init , trans, error, sname="z3"):

        self.variables = variables       # FOTS variables
        self.init  = init                # FOTS init as unary predicate in "variables"
        self.error = error               # FOTS error condition as unary predicate in "variables"
```

```
        self.trans = trans                # FOTS transition relation as a binary transition relation
                                          # in "variables" and "prime variables"

        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [self.error]       # inializa com uma só frame: a situação de error

        self.solver = ps.Solver(name=sname)
        self.solver.add_assertion(self.init)     # adiciona o estado inicial como uma asserção sempre presente

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        T = self.trans.substitute(dict(zip(self.prime_variables,freshs)))
        F = self.frames[-1].substitute(dict(zip(self.variables,freshs)))
        self.frames.append(ps.Exists(freshs, ps.And(T, F)))

    def unroll(self,bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: tentativas ultrapassam o limite %d "%bound)
                break
            elif self.solver.solve(self.frames):
                self.new_frame()
                n += 1
            else:
                print("sucesso: tentativa %d "%n)
                break

class Cycle(EPU):
    def __init__(self,variables,pre,pos,control,body,sname="z3"):
        init  = pre
        trans = ps.And(control,body)
        error = ps.Or(control,ps.Not(pos))
        super().__init__(variables, init, trans, error, sname)



bits = 16
N = ps.BV(((2**bits)-1), bits)
zero = ps.BVZero(bits)
one  = ps.BVOne(bits)

# O ciclo
m16 = ps.Symbol("m16", pt.BV16)
n16 = ps.Symbol("n16", pt.BV16)
x16 = ps.Symbol("x16", pt.BV16)
y16 = ps.Symbol("y16", pt.BV16)
r16 = ps.Symbol("r16", pt.BV16)

variables = [m16, n16, x16, y16, r16]


pre = ps.And(n16 < N, m16 < N, m16 >= zero, n16 >= zero, r16.Equals(zero), x16.Equals(m16), y16.Equals(n16))
pos = r16.Equals(m16 * n16)
cond = y16 > zero
```

```
ifBody = ps.And(
    ps.Implies(ps.Equals(ps.BVAnd(y16, one), one), ps.And(
        ps.Equals(prime(y16), ps.BVSub(y16, one)),
        ps.Equals(prime(x16), ps.BVAdd(r16, x16))
    )),
    ps.Implies(ps.Not(ps.Equals(ps.BVAnd(y16, one), one)), ps.And(
            ps.Equals(prime(y16), y16),
            ps.Equals(prime(x16), x16)
    ))
)

trans = ps.And(
    ifBody,
    ps.Equals(prime(x16), ps.BVLShl(x16, one)),
    ps.Equals(prime(y16), ps.BVLShr(y16, one))
)


W = Cycle(variables,pre,pos,cond,trans)
W.unroll(bits)
```

```
sucesso: tentativa 2
```