TP4_com_havoc

January 17, 2021

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

0.1 Modelação de programas com FOTS

Um programa pode ser modelado por um FOTS da seguinte forma: - O estado é constituído pelas variáveis do programa mais uma variável para o respectivo *program counter* - Os estados iniciais são caracterizados implicitamente por um predicado sobre as variáveis de estado - As transições são caracterizadas implicitamente por um predicado sobre pares de estados

Considerando o seguinte programa:

```
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
```

Queremos:

0.2 1. Provar por indução que o programa termina

Vamos começar por definir as funções declare, init e trans.

Função declare

[2]:
```
def declare(i):
    trace = {}
    trace["x"] = BitVec('x_'+str(i), 16)
    trace["y"] = BitVec('y_'+str(i), 16)
    trace["r"] = BitVec('r_'+str(i), 16)
    trace["m"] = BitVec('m_'+str(i), 16)
    trace["n"] = BitVec('n_'+str(i), 16)
    trace["pc"] = BitVec('pc_'+str(i), 16)

    return trace
```

[3]:
```
declare(0)
```

[3]: {'x': x_0, 'y': y_0, 'r': r_0, 'm': m_0, 'n': n_0, 'pc': pc_0}

Função init O estado inicial é caracterizado pelo seguinte predicado:

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

[4]:
```
def init(trace):
    return And(trace["pc"]==0,trace["r"]==0, trace["m"]>=0, trace["n"]>=0,␣
 ↪trace["x"]==trace["m"], trace["y"]==trace["n"])
```

[5]:
```
init(declare(0))
```

[5]:
```
And(pc_0 == 0,
    r_0 == 0,
    m_0 >= 0,
    n_0 >= 0,
    x_0 == m_0,
    y_0 == n_0)
```

Função trans As transições possíveis no FOTS são caracterizadas pelo seguinte predicado:

$$[(pc = 0 \wedge pc' = 1 \wedge$$

$$\vee(pc = 1 \wedge y\%1 = 0 \wedge pc' = 2 \wedge m' = m \wedge n' = n \wedge r' = r + x \wedge x' = x \wedge y' = y - 1) \vee (pc = 1 \wedge y\&! = 1 \wedge pc' =$$

[6]:
```
def trans(atual, prox):
    # Condições para pc == 0
    # y > 0:
```

```
    ciclo1_pc0 = And(atual["pc"]==0,prox["pc"]==1,atual["y"]>0,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"], prox["x"]==atual["x"], prox["y"]==atual["y"])

    # y <=0
    ciclo2_pc0 = And(atual["pc"]==0,prox["pc"]==3,atual["y"]<=0,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"], prox["x"]==atual["x"], prox["y"]==atual["y"])

    pc0 = Or(ciclo1_pc0, ciclo2_pc0)

    # Condições para pc == 1
    # if if y & 1 == 1:
    if1_pc1 = And(atual["pc"]==1,atual["y"]&1==1,prox["pc"]==2,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"]+atual["x"], prox["x"]==atual["x"], prox["y"]==atual["y"]-1)

    # if if y & 1 =! 1:
    if2_pc1 = And(atual["pc"]==1,atual["y"]&1!=1,prox["pc"]==2,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"], prox["x"]==atual["x"], prox["y"]==atual["y"])

    pc1 = Or(if1_pc1, if2_pc1)

    # Condições para pc = 2
    pc2 = And(atual["pc"]==2,prox["pc"]==0,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"],prox["x"]==atual["x"]<<1, prox["y"]==atual["y"]>>1)

    # Condições para pc = 3 (ciclo termina)
    pc3 = And(atual["pc"]==3,prox["pc"]==atual["pc"], atual["y"]<=0,prox["m"]==atual["m"], prox["n"]==atual["n"], prox["r"]==atual["r"], prox["x"]==atual["x"], prox["y"]==atual["y"])

    return Or(pc0, pc1, pc2, pc3)
```

Utilizando indução com um lookahead de l queremos provar, para um dado traço $s = \{s_i \mid i = 0, 1, \ldots, k-1\}$ de um FOTS, que o programa termina - a variável pc toma o valor 3.

Temos que descobrir, então, um *variante* V que satisfaz as seguintes condições:

  - O variante é sempre positivo, ou seja, \$ V(s) $\geq$ 0\$*Ovariantedescrescesempre*(*estritamente*)*ouatingeovalor*0, *ou* $(V(s') < V(s) \vee V(s') = 0))$\$
- Quando o variante é 0 verifica-se necessariamente $pc = 3$, ou seja, \$ (V(s)=0 $\rightarrow pc = 3)$\$

O variante pode ser definido como:

$$V(s) \equiv y_s$$

Usando induçao com lookahead:

[7]:
```
def induction_base(declare, init, var , prop, k):
    s = Solver()
    trace = {i: declare(i) for i in range(2)}

    s.add(init(trace[0]))
    s.add(Not(var(trace[0], trans, k)))

    if s.check() == sat:
        print(f'A propriedade ({prop}) no estado inicial abaixo descrito falhou')
        m = s.model()
        for v in trace[0]:
            print(v, "=", m[trace[0][v]])
        return False
    else:
        print(f'A propriedade ({prop}) é válida para o estado inicial ')
        return True


def induction_step(declare, trans, var , prop, k):
    s = Solver()
    trace = {i: declare(i) for i in range(2)}


    s.add(var(trace[0], trans, k))
    s.add(Not(var(trace[0], trans, k)))

    if s.check() == sat:
        print(f'Propriedade ({prop}) inválida no passo de indução para o traço␣
 ↪abaixo descrito')
        m = s.model()

        for v in trace[0]:
            print(v, "=", m[trace[0][v]])
        return False
    else:
        print(f'Propriedade ({prop}) é válida no passo de indução')
        return True
```

[8]:
```
def variant(trace):
    return trace["y"]

def var_positive(trace, trans, l):
    traces = {i: declare(i) for i in range(1, l+1)}
```

```
    c1 = And([trans(traces[i], traces[i+1]) for i in range(1, l)] +␣
 ↪[trans(trace, traces[1])])
    c2 = variant(traces[l])>=0
    r = ForAll(list(traces[l].values()), Implies(c1, c2))
    return r

def var_decreases(trace, trans, l):
    traces = {i: declare(i) for i in range(1, l+1)}
    c1 = And([trans(traces[i], traces[i+1]) for i in range(1, l)] +␣
 ↪[trans(trace, traces[1])])
    c2 = Or(variant(traces[l])<variant(trace), variant(traces[l])==0)
    r = ForAll(list(traces[l].values()), Implies(c1, c2))
    return r

def var_useful(trace, trans, l):
    traces = {i: declare(i) for i in range(1, l+1)}
    c1 = And([trans(traces[i], traces[i+1]) for i in range(1, l)] +␣
 ↪[trans(trace, traces[1])])
    c2 = Implies(variant(traces[l])==0, traces[l]["pc"]==3)
    r = ForAll(list(traces[l].values()), Implies(c1, c2))
    return r
```

[9]:
```
def induction_always(declare, init, trans, var , prop, k):
    return induction_base(declare, init, var , prop, k) and␣
 ↪induction_step(declare, trans, var, prop, k)
```

[10]:
```
induction_always(declare, init, trans, var_positive,"positive", 10)
```

```
A propriedade (positive) é válida para o estado inicial
Propriedade (positive) é válida no passo de indução
```

[10]: True

[11]:
```
induction_always(declare, init, trans, var_decreases,"decreases", 10)
```

```
A propriedade (decreases) é válida para o estado inicial
Propriedade (decreases) é válida no passo de indução
```

[11]: True

[12]:
```
induction_always(declare, init, trans, var_useful, "useful", 10)
```

```
A propriedade (useful) é válida para o estado inicial
Propriedade (useful) é válida no passo de indução
```

[12]: True

0.3 Correção Parcial

0.3.1 1. Havoc

A denotação do triplo de Hoare $\{\phi\}$while b do$\{\theta\}\, C\, \{\psi\}$, traduzido desta forma, permite garantir as propriedades de "inicialização", "preservação" e "utilidade" do invariante θ

$$[\,\text{assume } \phi\,;\text{assert } \theta\,;\text{havoc } \vec{x}\,;(\,(\text{assume } b \land \theta\,;\; C\,;\text{assert } \theta\,;\text{assume } \mathit{False}) \,||\, \text{assume } \neg b \land \theta\,)\,;\text{assert } \psi\,]$$
$$=$$
$$\phi \rightarrow \theta \land \forall \vec{x}.\,(\,(b \land \theta \rightarrow [C\,;\text{assert } \theta]) \land (\neg b \land \theta \rightarrow \psi)\,)$$

. Pré-condição: $m >= 0 \land n >= 0 \land r = 0 \land x = m \land y = n$

. Pós-condição: $r = m * n$

. Invariante: $x * y + r = m * n \land y >= 0$

. Condição (cond): $y\%1 == 1$

Também podemos extender a condição: \$ [C;; {assert}; θ]\$

\$ [C;; {assert}; θ] = $[(C\_1||C\_2); \{\text{assert}\}; `] = [\text{C}\_1; ; \{\text{assert}\}; `]; \land; [\text{C}\_2; ; \{\text{assert}\}; `] = (\text{cond} \rightarrow `[\text{y}/\text{y} >> 1][\text{x}/\text{x} << 1][\text{r}/\text{r} + \text{x}][\text{y}/\text{y} - 1]); \land; (\neg\text{cond} \rightarrow `[\text{y}/\text{y} >> 1][\text{x}/\text{x} << 1])$\$

[13]:
```python
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])
```

[14]:
```python
def havoc(nbits):
    m, n, r, x, y = BitVecs("m n r x y", nbits)

    preC = And(m>=0,n>=0,r==0,x==m,y==n)
    posC = r==m*n
    inv = And(x*y+r==m*n,y>=0)
    b = y>0
    cond = y&1==1

    # INICIALIZAÇÃO:
    init = inv

    # UTILIDADE:
    # condiçoes do [C; assert inv]:
```

```
    cond1 =␣
↪Implies(cond,substitute(substitute(substitute(substitute(inv,(y,y>>1)),(x,x<<1)),(r,r+x)),(y,
    cond2 = Implies(Not(cond), substitute(substitute(inv,(y,y>>1)),(x,x<<1)))

    # [C; assert inv]:
    cAssertInv = And(cond1,cond2)

    # (b^inv -> cAsssertInv)
    util = Implies(And(b,inv),cAssertInv)

    # PRESERVAÇÃO:
    pres = Implies(And(Not(b),inv),posC)

    wpc=Implies(preC, And(init, ForAll([x,y,r], And(util, pres))))

    prove(wpc)
```

[15]:
```
havoc(4)
```

```
Proved
```

[16]:
```
havoc(8)
```

```
Proved
```

[17]:
```
havoc(10)
```

```
Proved
```