vakahxton

November 23, 2023

1 TP3 - Algortimo estendido de Euclides

1.0.1 Exercício 2 (Prova de Correção)

Grupo 05

`Eduardo André Silva Cunha A98980`
`Gonçalo Emanuel Ferreira Magalhães A100084`

2 Problema

Este exercício é dirigido à prova de correção do algoritmo estendido de Euclides

1. Construa a asserção lógica que representa a pós-condição do algoritmo. Note que a definição da função
   gcd é $\gcd(a, b) \;\equiv\; \min\{\, r > 0 \,|\, \exists\, s, t \,\centerdot\, r = a * s + b * t \,\}$.
2. Usando a metodologia do comando havoc para o ciclo, escreva o programa na linguagem dos comandos anotados (LPA). Codifique a pós-condição do algoritmo com um comando assert.
3. Construa codificações do programa LPA através de transformadores de predicados: "weakest pre-condition" e "strongest post-condition".
4. Prove a correção do programa LPA em ambas as codificações.

Imports

[1]:
```
from pysmt.shortcuts import *
from pysmt.typing import *
#from z3 import *
```

Variáveis

[2]:
```
global n_global
n_global = 10

global a_global
a_global = 48

global b_global
b_global = 18

global N_global
```

```
N_global = 50


r = Symbol('r', INT)
r_linha = Symbol('r_linha', INT)
s = Symbol('s', INT)
s_linha = Symbol('s_linha', INT)
t = Symbol('t', INT)
t_linha = Symbol('t_linha', INT)
```

Construa a asserção lógica que representa a pós-condição do algoritmo. Note que a definição da função gdc é

- $\gcd(a, b) \equiv \min\{r > 0 \,|\, \exists\, s, t \,.\, r = a * s + b * t\}$.

[3]:

```
def pos_cond(output_r, output_s, output_t): # Recebe os 3 valores finais
    with Solver(name="z3") as s:            # Vamos utilizar um solver z3 para a verificação do "min"

        r_novo = Symbol('r_novo',INT)       # cria novas variaveis
        s_novo = Symbol('s_novo',INT)
        t_novo = Symbol('t_novo',INT)

        eq = Equals(r_novo, Int(a_global) * s_novo + Int(b_global) * t_novo) # lei de Bezout

        # Restrições
        s.add_assertion(r_novo > 0)         # r_novo tem de ser um resultado positivo
        s.add_assertion(r_novo < output_r)  # r_novo tem de ser um resultado menor que o r final
        s.add_assertion(eq)                 # os valores tem de respeitar sempre bezout

        if s.solve():       # se resolver é porque encontrou um r_novo menor que o r, que satisfaz bezout
            print("Resposta do modelo:", s.get_model())
            return False # então retorna falso

    # Caso não encontre outra resposta, retorna se os valores dados respeitam bezout
    return Equals(output_r,(Int(a_global) * output_s + Int(b_global) * output_t))
```

Usando a metodologia do comando havoc para o ciclo, escreva o programa na linguagem dos comandos anotados (LPA). Codifique a pós-condição do algoritmo com

um comando assert.

- Função dada nas aulas para auxiliar nos métodos de prova

[7]:
```
def prove(f):
    with Solver(name="z3") as s:
        s.add_assertion(Not(f))
        if s.solve():
            print("Failed to prove.")
        else:
            print("Proved.")
```

[11]:
```
# ax1, garante os valores iniciais das variaveis
ax1 = And(Equals(r, Int(a_global)), Equals(r_linha, Int(b_global)), Equals(s,␣
 ↪Int(1)), Equals(s_linha, Int(0)), Equals(t, Int(0)), Equals(t_linha, Int(1)))

# ax2, garante o passo do ciclo while
ax2 = And(
    Implies(NotEquals(r_linha, Int(0)), And(
        Equals(r_linha, r - (r_linha / r) * r_linha),
        Equals(s_linha, s - (r_linha / r) * s_linha),
        Equals(t_linha, t - (r_linha / r) * t_linha),
        Equals(r, r_linha),
        Equals(s, s_linha),
        Equals(t, t_linha)
    ))
)

#print(pos_cond(r,s,t))
#assert(pos_cond(r,s,t)) --> erro

axioms = And(ax1,ax2)

#prove(Implies(axioms,pos_cond(r,s,t))) --> erro
prove(Implies(axioms,And(Equals(r, Int(6)), Equals(s, Int(-1)), Equals(t,␣
 ↪Int(3)))))
```

```
Proved.
```

Construa codificações do programa LPA através de transformadores de predicados: "weakest pre-condition" e "strongest post-condition".

[5]:
```
# Novos simbolos necessários
a = Symbol('a', INT)
b = Symbol('b', INT)
N = Symbol('N', INT)

# atribuição dos valores globais às novas variaveis
```

```
pre_atrib =
↪And(Equals(a,Int(a_global)),Equals(b,Int(b_global)),Equals(N,Int(N_global)))

# assume
pre = And(a > 0, b > 0, a < N, b < N)

# pos cond
pos = And(Equals(r_linha, Int(0)), Equals(r,(Int(a_global) * s + Int(b_global)
↪* t)))

# invariante, lei de bezout
inv = Equals(r, a_global * s + b_global * t)

# Init, atribuição dos valores iniciais
ini = substitute(inv, {r: Int(a_global), r_linha: Int(b_global), s: Int(1),
↪s_linha: Int(0), t: Int(0), t_linha: Int(1)})

# pres, atribuição dos novos valores dentro do ciclo while
pres = Implies(NotEquals(r_linha, Int(0)),
               substitute(inv,
                          {r_linha: r - (r_linha / r) * r_linha,
                           s_linha: s- (r_linha / r) * s_linha,
                           t_linha: t - (r_linha / r) * t_linha,
                           r: r_linha,
                           s: s_linha,
                           t: t_linha})
               )
# util, garante o invariante durante o ciclo
util = Implies(r_linha >= 0, inv)


td = And(axioms,pre_atrib,pre,pos,inv,ini,pres,util)
# Prova do teorema estendido de Euclides
prove(Implies(td,And(Equals(r, Int(6)), Equals(s, Int(-8)), Equals(t, Int(3)))))
```

```
Proved.
```

Prove a correção do programa LPA em ambas as codificações.

[6]:
```
# "weakest pre-condition"
WPC = Implies(pre, And(ini, ForAll([r,s,t],And(pres,util))))
prove(Implies(td,WPC))

# "strongest post-condition"
SPC = And(pre, Implies(ini, ForAll([r,s,t],And(pres,util))))
prove(Implies(td,SPC))
```

```
Proved.
```

Proved.