

# TRABALHO 2 - ESPECIFICAÇÃO FORMAL DE SOFTWARE

Carlos Diego N. Damasceno - 9055752, Universidade de São Paulo

30/10/2015

## Problema 1

Escreva uma função `triang` que recebe três valores reais positivos e retorna um inteiro que representa se os valores formam um triângulo e qual o seu tipo. Deve retornar -1 se não for um triângulo válido, 0 se for equilátero, 1 se for isósceles e 2 se for escaleno. Use o z3 para provar que a função está correta.

Algoritmo 1: Função **triang** que dado três valores retorna o tipo de triângulo.

```
1  public static int triang(double a, double b, double c) {
2      if (
3          !(((a+b)>c)||((b+c)>a)||((c+a)>b)||((a>0)||((b>0)||((c>0)))
4      ){
5          return -1; // not valid
6      }else if (
7          ((a == b) && (b ==c)) // equilateral
8      ){
9          return 0;
10     }else if (
11         && ((a == b) ^ (a == c) ^ (b == c)) // isosceles
12     ){
13         return 1;
14     }else {
15         return 2;
16     }
17     return -1;
18 }
```

A partir do programa apresentado no Algoritmo 1, foi especificado o seguinte código em Z3 que prova a sua corretude. O código em Z3 que prova a corretude do Algoritmo 1 pode ser visualizado em Algoritmo 2. Entre as linhas 5-8 temos a assertiva que verifica casos onde um triângulo inválido é recebido (valores *não positivos* e onde *a soma de dois lados não é maior que o terceiro*). Entre as linhas 9-13 temos definida a assertiva que verifica triângulos equiláteros, ou seja, casos onde o triângulo é válido (linhas 10-11) e tem todos os lados iguais (linha 13). Entre as linhas 14-19 temos a assertiva responsável por verificar se o triângulo é válido, não equilátero e isósceles. Entre as linhas 20-25 temos a assertiva que verifica a validade do triângulo e a negação dos demais tipos de triângulos, restando apenas o caso

onde todos os lados são diferentes, ou seja escaleno.

Algoritmo 2: Especificação formal em Z3 da função **triang**.

---

```
1 (declare-fun a () Real)
2 (declare-fun b () Real)
3 (declare-fun c () Real)
4 (declare-fun triang () Int)
5 (assert (or (or
6             (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
7             (> a 0) (> b 0) (> c 0))
8             (= triang -1))) ; if( not valid ) return -1
9 (assert (or (not (or
10             (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
11             (> a 0) (> b 0) (> c 0)))
12             (not (and (= a b) (= a c))) ;; a==b && b==c (equilateral)
13             (= triang 0))) ; if( equilateral ) return 0
14 (assert (or (not(or
15             (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
16             (> a 0) (> b 0) (> c 0)))
17             (and (= a b) (= a c)) ;; a==b && b==c (equilateral)
18             (not (xor (= a b) (= a c) (= b c))) ;; (isosceles)
19             (= triang 1))) ; if( isosceles ) return 1
20 (assert (or (not(or
21             (> (+ a b) c) (> (+ b c) a) (> (+ a c) b)
22             (> a 0) (> b 0) (> c 0)))
23             (and (= a b) (= a c)) ;; a==b && b==c (equilateral)
24             (xor (= a b) (= a c) (= b c)) ;; (scalene)
25             (= triang 2))) ; if( scalene ) return 2
26 ;(assert (and (not (= a b c)) (= triang 0))) ; equilateral
27 ;(assert (and
28 ;    (not (xor (= a b) (= a c) (= b c)))
29 ;    (= triang 1) )) ; isosceles
30 ;(assert (and (or (= a b) (= a c) (= b c)) (= triang 2) )) ; scalene
31 (check-sat)
32 (get-model)
```

---

Nas linhas 26, 27-29 e 30 temos, respectivamente as assertivas que verificam se um triângulo é equilátero, isósceles ou escaleno. E por fim, nas linhas 31 e 32 temos a chamada das operações de checagem de satisfabilidade e visualização do modelo, se houver.

## Problema 2

Escreva uma função que calcula o MDC de dois números usando o algoritmo de Euclides. Escreva outra função que recebe três números e calcula o MDC desses números usando a função anterior. Use o z3 para gerar casos de teste (ou seja, triplas de números) que satisfaça aos critérios:

- a) Todos-nós
- b) Todos-usos

Algoritmo 3: Funções que calculam MDC para dois e três valores inteiros usando algoritmo de euclides recursivo.

---

```
1    public int gcd(int a, int b, int c) {
2        return gcd(a,gcd(b,c));
3    }
4
5    public int gcd(int a, int b) {
6        if(b==0) {
7            return a;
8        }
9        else {
10           return gcd(b, a % b);
11        }
12    }
```

---

Os métodos em Java apresentado no Algoritmo 3 permitem calcular o MDC de dois e três números inteiros. Para gerar o MCD de três valores inteiros, o calcula o MDC de um dos três valores com o MDC dos dois valores restante. O MDC de um par de valores inteiros, por sua vez, é calculado recursivamente usando o método de Euclides.

Para gerar testes que atendam aos critérios *Todos-Usos* e *Todos-Nós*, o trecho do Algoritmo 3 onde a função `gcd(int a, int b)` é chamada dentro da função `gcd(int a, int b, int c)` (Linha 2 do Algoritmo 3) foi modelado como apresentado no Algoritmo 4.

Entre as linhas 10-19 do Algoritmo 4 foram definidas restrições que garantem que o teste gerado cobrirá o uso predicativo da variável 'b' na linha 6 e a linha 7 do Algoritmo 3. Entre as linhas 21-30 do Algoritmo 4 o mesmo procedimento anteriormente é feito, entretanto, a fim de garantir a cobertura da linha 10 do Algoritmo 3.

Em conjunto, as assertivas apresentadas anteriormente permitem atender os dois critérios de teste estabelecidos para a geração. Ao ser executado no Z3 o código acima gerou a tripla de dados de entrada  $a = 0$ ,  $b = 1$  e  $c = 0$ . Um trecho da saída obtida a partir da execução do código em Z3 em questão pode ser visto no Algoritmo 5

Algoritmo 4: Especificação Formal que permite gerar casos de teste que atendem *Todos-Usos* e *Todos-Nós*.

---

```
1 (declare-const a Int) ; a value
2 (declare-const b Int) ; b value
3 (declare-const c Int) ; c value
4 (declare-fun gcd (Int Int) Int)
5 (declare-fun gcd3 (Int Int Int) Int)
6
7 ; gcd(a,b,c) == gcd(a,gcd(b,c))
8 (assert (= (gcd3 a b c) (gcd a (gcd b c)) ) )
9
10 ; REFERS TO gcd(b,c)
11 (declare-const a1 Int)
12 (declare-const b1 Int)
13 (assert (and (= a1 b) (= b1 c)))
14 (declare-const ifb0 Bool)
15 (assert
16   (ite (= b1 0)
17     (and (= ifb0 true) (= a1 (gcd a1 b1)))
18     (and (= ifb0 false) (= (gcd a1 b1) (gcd b1 (mod a1 b1))))))
19 (assert (= ifb0 true)) ; covers "return a" coverage
20
21 ; REFERS TO gcd(a,gcd(b,c))
22 (declare-const a2 Int)
23 (declare-const b2 Int)
24 (assert (and (= a2 a) (= b2 (gcd a1 b1))))
25 (declare-const ifb1 Bool)
26 (assert
27   (ite (= b2 0)
28     (and (= ifb1 true) (= a2 (gcd a2 b2)))
29     (and (= ifb1 false) (= (gcd a2 b2) (gcd b2 (mod a2 b2))))))
30 (assert (= ifb1 false)) ; covers "gcd(a,gcd(b,c))" coverage
31
32 (assert (and ; assures gcd is divisor
33   (= 0 (mod b (gcd b c)))
34   (= 0 (mod c (gcd b c)))
35   (= 0 (mod a (gcd3 a b c)))
36   (= 0 (mod b (gcd3 a b c)))
37   (= 0 (mod c (gcd3 a b c)))))
38
39 (check-sat)
40 (get-model)
```

---

Algoritmo 5: Especificação Formal que permite gerar casos de teste que atendem *Todos-Usos* e *Todos-Nós*.

---

```
1
2 sat
3 (model
4   ...
5   (define-fun b () Int
6     1)
7   (define-fun a () Int
8     0)
9   ...
10  (define-fun c () Int
11    0)
12  ...
13 )
```

---

## Considerações Finais

Os dados referentes ao trabalho podem ser encontrados NO diretório *efs\_trabalho02\_z3* no seguinte link: [https://github.com/damascenodiego/formalSpecification\\_usp\\_2015/](https://github.com/damascenodiego/formalSpecification_usp_2015/).