# CS:5810 Formal Methods in Software Engineering

#### Recursion and Termination

Copyright 2020-25, Graeme Smith and Cesare Tinelli.

Produced by Cesare Tinelli at the University of Iowa from notes originally developed by Graeme Smith at the University of Queensland. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

### Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x
  if x == 0
   d := 0;
  } else {
   var y;
    y := Double(x - 1);
    d := y + 2;
```

#### Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x { x >= 0 ==> (x != 0 ==> x > 0) }
                          \{ x != 0 ==> x > 0 \}
  if x == 0
                       \{ (x == 0 ==> 0 == 2*x) \&\&
                            (x != 0 ==> x - 1 >= 0)
                          \{ 0 == 2*x \}
   d := 0;
                          \{ d == 2*x \}
  } else {
                          { forall y :: x - 1 >= 0 && true}
                          \{ x - 1 >= 0 \&\& forall r :: r == 2*(x - 1) ==> \}
    var y;
                                           r + 2 == 2*x
    y := Double(x - 1); { y + 2 == 2*x }
    d := y + 2; { d == 2*x }
                       \{ d == 2*x \}
```

### Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x
  if x == 0
                               Recursive methods can be analyzed
                               like any methods that call other methods ...
   d := 0;
  } else {
                               ... if they terminate!
    var y;
    y := Double(x - 1);
    d := y + 2;
```

### Problematic recursion

```
method BadDouble(x: int) returns (d: int)
  ensures d == 2*x
 var y := BadDouble(x - 1);
                                              Does not terminate!
  d := y + 2;
method BadIdentity(x: int) returns (y: int)
  ensures y == x
  if x % 2 == 2
                                              Does not terminate!
   \{ y := x; \}
  else
    { y := BadIdentity(x); }
```

### Fibonacci function

```
nat: non-negative integers
 function Fib(n: nat): nat {
    if n < 2 then n else Fib(n - 2) + Fib(n - 1) }</pre>
                              Fib(701)
Terminates!
                                             Fib(700)
               Fib(699)
                                     Fib(698)
        Fib(697)
                                                     Fib(699)
                      Fib(698)
```

## How to prove termination?

```
function Fib(n: nat): nat
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
function Ack(m: nat, n: nat): nat
                                          Also terminates!
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
       else Ack(m - 1, Ack(m, n - 1))
```

#### Termination metric

```
function Fib(n: nat): nat

    Suggestion for Dafny

  decreases n 4
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
function SeqSum(s: seq<int>, lo: nat, hi: nat): nat
  requires 0 <= lo <= hi <= |s|
  decreases hi - lo
  if lo == hi then 0 else s[lo] + SeqSum(s, lo + 1, hi)
```

#### Termination metric

Termination metrics do not have to be natural numbers

Any set of values with a well-founded order can be used

An order > is well-founded when

- > is *irreflexive*: a > a never holds
- > is *transitive*: if a > b and b > c then a > c
- there is no infinite descending chain:  $a_0 > a_1 > a_2 > ...$

# Well-founded orders in Dafny

type	X ≻ Y ("X decreases to Y")  iff	
bool	X && !Y	true decreases to false
int	X > Y && X >= 0	negative ints not ordered
real	X - 1.0 >= Y && X >= 0.0	
set <t></t>	X is a proper superset of Y	⊃, not ⊇
seq <t></t>	X strictly contains Y	e.g., [a, b, c] > [b, c]
datatypes	X structurally includes Y	e.g., ((a, b), (c, d)) > (a, b)

## Lexicographic tuples

A *lexicographic order* orders tuples of values

It does component-wise comparison, where earlier components are more significant

#### **Examples:**

```
4, 12 > 4, 11 > 4, 2 > 3, 5260 > 2, 0
4, 12 > 4, 12, 365, 0
12, true, 1.9 > 12, false, 57.3
```

## Lexicographic tuples

A *lexicographic order* orders tuples of values

It does component-wise comparison, where earlier components are more significant

**Theorem:** A lexicographic order is well founded whenever all the component orders are well founded

# Remaining study

The following method simulates your time until graduation, from when you have h hours of study left in course c

```
method Study(c: nat, h: nat)
  decreases c, h
                                  -c, h > c, h-1
  if h != 0 { Study(c, h - 1); }
  else if c == ∅ { } // graduation!
  else { var h1 := ReqStudyTime(c - 1);
         Study(c - 1, h1);
                                 c, h > c-1, h1
```

#### Ackermann function

```
function Ack(m: nat, n: nat): nat
 decreases m, n
  if m == 0 then n + 1
 else if n == 0 then Ack(m - 1, 1)
       else Ack(m - 1, Ack(m, n - 1))
```

## Mutually recursive functions

```
method StudyPlan(c: nat)
  requires c <= 40
                                             40-c > 40-c, h
  decreases 40 - c
 if c != 40 { var h := ReqStudyTime(c); Learn(c, h); }
                              40-c, h > 40-(c+1)
method Learn(c: nat, h: nat)
  requires c < 40
                                         40-c, h > 40-c, h-1
  decreases 40 - c, h
  if h == 0 { StudyPlan(c + 1); } else { Learn(c, h - 1); }
```