The role of termination in program verification

K. Rustan M. Leino

18-20 March 2026
PhD Open, U Warsaw
Warsaw, Poland

Goal of lectures

To review basics of program verification, and
to give a perspective on

induction / well-founded recursion

as a common thread in

programming, function definitions, and proofs

Take-home message:

- You can call whatever you want whenever you want, provided
  - you satisfy the precondition, and
  - you can show a decrease in some fixed well-founded order

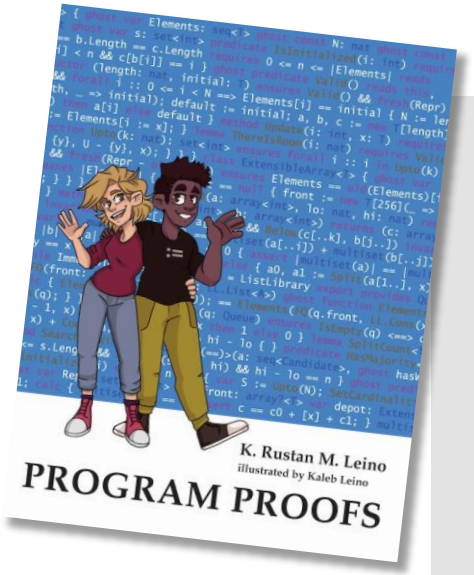
Vehicle for our journey

- Dafny (`dafny.org`)
- Programming language
  - General purpose
  - Verification-aware (proof-oriented)
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs
- VS Code integration
- AI friendly
  - LLMs do well with Dafny

Some uses of Dafny

- Teaching
  - Over a decade, many universities
  - Book (MIT Press): `program-proofs.com`
- At AWS
  - AWS Encryption SDK
  - AWS Database Encryption SDK
  - AWS's authorization engine

    Paper at ICSE 2025.
    Talk at AWS re:Inforce 2024:
    https://youtu.be/oshxAJGrwMU
  - …
- Research projects
- …



CombineSplit.dfy

Methods and pre-/post-conditions

```
method Split(c: int) returns (a: int, b: int)
  requires 0 <= c < 10_000
  ensures 0 <= a < 100 && 0 <= b < 100
  ensures c == 100 * a + b
{
  a := c / 100;
  b := c % 100;
}

method Combine(a: int, b: int) returns (c: int)
  requires 0 <= a < 100 && 0 <= b < 100
  ensures 0 <= c < 10_000
  ensures c == 100 * a + b
{
  if a == 0 {
    c := b;
  } else if a == 1 {
    c := 100 + b;
  } else {
    c := 100 * a + b;
  }
}

method Caller(c: int)
  requires 0 <= c < 10_000
{
  var a, b := Split(c);
  var d := Combine(a, b);
  assert c == d;
}
```

Loop invariants

```
method Double(x: int) returns (r: int)
  ensures r == 2 * x
{
  r := x + x;
}

method Double'(x: int) returns (r: int)
  ensures r == 2 * x
{
  if x < 0 {
    r := x + x;
  } else {
    var y := 3 * x;
    r := y - x;
  }
}
```

Double.dfy 1/2

Loop invariants

```
method Double''(x: nat) returns (r: int)
  ensures r == 2 * x
{
  if x == 0 {
    return 0;
  } else {
    r := Double''(x - 1);
    assert r == 2 * (x - 1) == 2 * x - 2;
    r := r + 2;
  }
}

method Double'3(x: nat) returns (r: int)
  ensures r == 2 * x
{
  r := 0;
  var i := 0;
  while i < x
    invariant 0 <= i <= x
    invariant 0 <= r
    invariant r == 2 * i
  {
    r := r + 2;
    i := i + 1;
  }
  assert i == x;
}
```

Min.dfy

Interacting with the verifier

```
method Min(s: seq<int>) returns (m: int)
  requires |s| != 0
  ensures m in s
  ensures forall j :: 0 <= j < |s| ==> m <= s[j]
{
  m := s[0];
  for i := 1 to |s|
    invariant m in s
    invariant forall j :: 0 <= j < i ==> m <= s[j]
  {
    if s[i] < m {
      m := s[i];
    }
  }
}
```

Binary search

```
method BinarySearch(a: array<int>, key: int) returns (r: int)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures 0 <= r ==> r < a.Length && a[r] == key
  ensures r < 0 ==> key !in a[..]
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 <= lo <= hi <= a.Length
    invariant key !in a[..lo] && key !in a[hi..]
  {
    var mid := (lo + hi) / 2;
    if a[mid] == key {
      return mid;
    } else if key < a[mid] {
      hi := mid;
    } else {
      lo := mid + 1;
    }
  }
  return -1;
}
```

Triangle numbers

```
function Triangle(n: nat): nat {
  if n == 0 then 0 else Triangle(n - 1) + n
}
```

$$T_n = \sum_{k=0}^{n} k$$

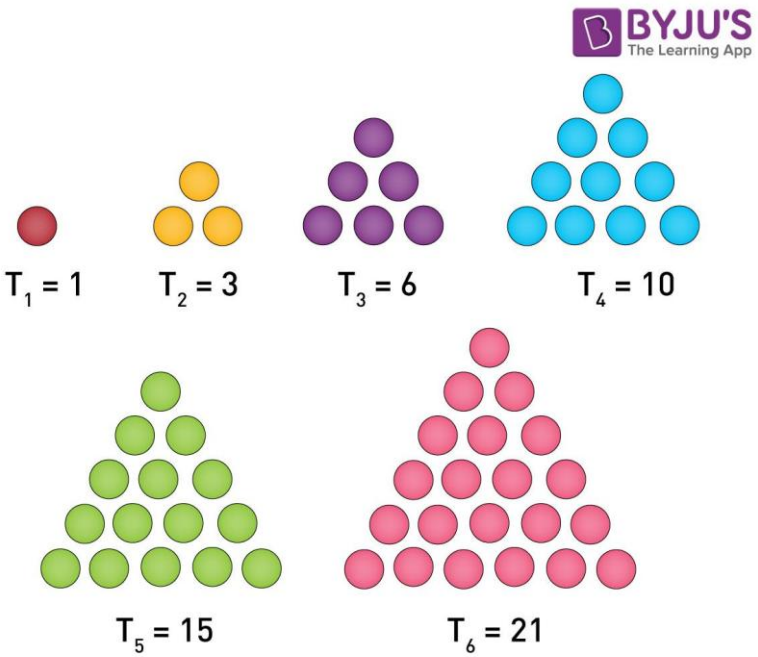


Image credit: byjus.com

Methods vs functions

```
method M(x: int) returns (y: int) {
  <statements>
}
```

- Can mutate state
- Can be nondeterministic
- Opaque (reasoning uses specification, never the body)

```
function F(x: int): int {
  <expr>
}
```

- Cannot mutate state
- Deterministic
- Transparent (reasoning uses body) – default or opaque

Statements vs expressions

statement	expression

`if G {`
`  <statements>`
`} else {`
`  <statements>`
`}`
optional
	`if G then <expr> else <expr>`

`match E`
`case Nil =>`
`  <statements>`
`case Cons(x, tail) =>`
`  <statements>`
	`match E`
	`case Nil =>`
	`  <expr>`
	`case Cons(x, tail) =>`
	`  <expr>`

Methods and lemmas

```
method Gauss(n: nat) returns (t: nat)
  ensures t == Triangle(n)

{
  t := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant t == Triangle(i)

  {
    i := i + 1;
    t := t + i;
  }
}
```

Methods and lemmas

```
method Gauss(n: nat) returns (t: nat)
  ensures t == Triangle(n)
  ensures t == n * (n + 1) / 2
{
  t := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant t == Triangle(i)
    invariant t == i * (i + 1) / 2
  {
    i := i + 1;
    t := t + i;
  }
}
```

Methods and lemmas

```
method Gauss(n: nat) returns (t: nat)
  ensures t == Triangle(n)
  ensures Triangle(n) == n * (n + 1) / 2
{
  t := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant t == Triangle(i)
    invariant Triangle(i) == i * (i + 1) / 2
  {
    i := i + 1;
    t := t + i;
  }
}
```

Methods and lemmas

```
method Gauss(n: nat)
  ensures Triangle(n) == n * (n + 1) / 2
{
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant Triangle(i) == i * (i + 1) / 2
  {
    i := i + 1;
  }
}
```

Methods and lemmas

```
lemma Gauss(n: nat)

  ensures Triangle(n) == n * (n + 1) / 2
{

  var i := 0;
  while i < n
    invariant 0 <= i <= n

    invariant Triangle(i) == i * (i + 1) / 2
  {
    i := i + 1;

  }
}
```

lemma ≈ method

- Stating and proving a lemma is just another application of program logic
  (Note: not based on Curry-Howard correspondence)
- For both methods and lemmas, you have to establish the postcondition for every input and every control path
- Calling a method/lemma obtains the information from its postcondition (and, for methods, experiences any side effects)

- Methods that call themselves are known as *recursive*
- Lemmas that call themselves are known as *inductive*

These are really the same!

The need for termination:

methods

```
method M(x: int)
  ensures P(x)
{
  M(x);
}

method Caller(x: int)
{
  M(x);
  assert P(x);
  …
}
```

The need for termination:

lemmas

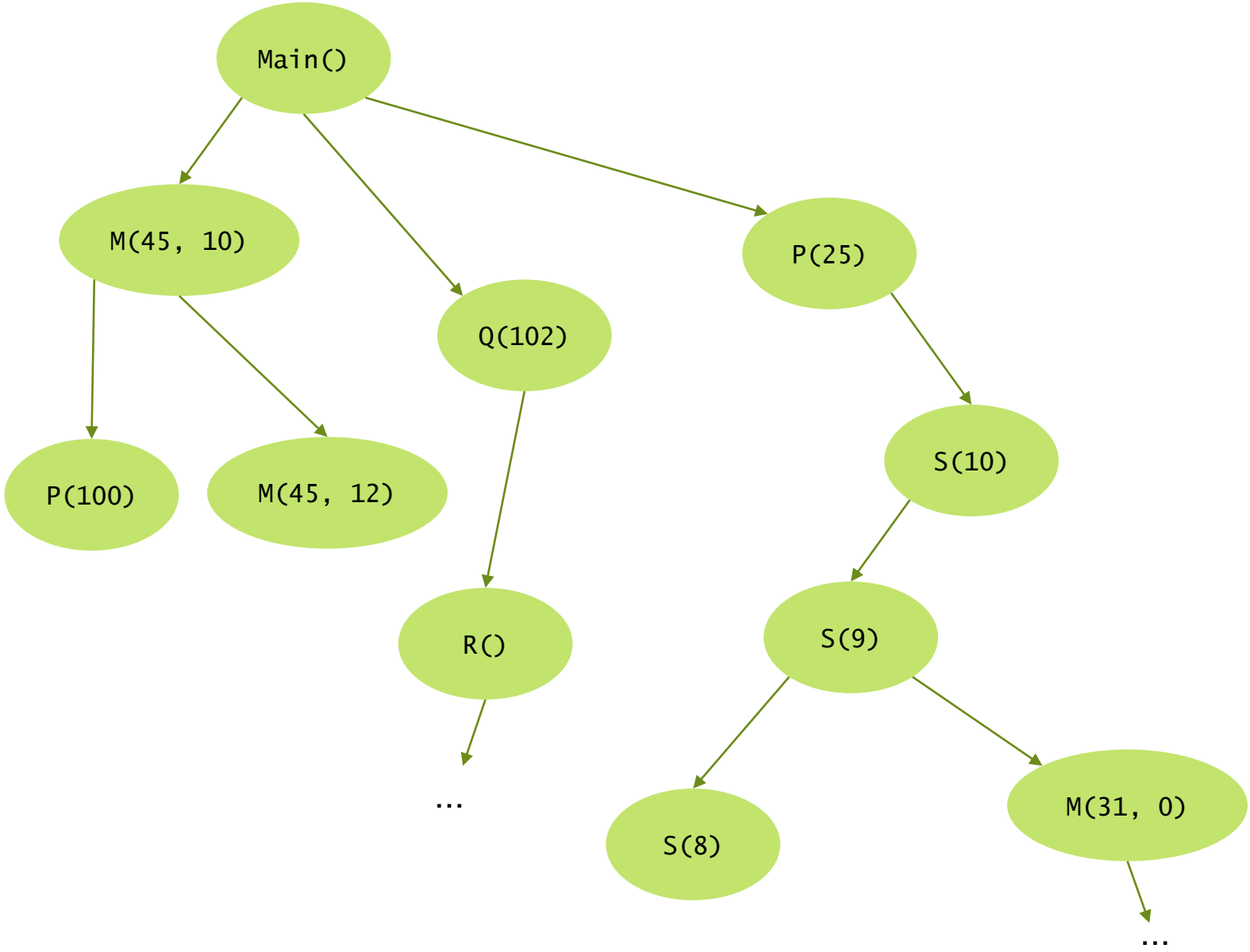
```
lemma L(x: int)
  ensures P(x)
{
  L(x);
}

method Caller(x: int)
{
  L(x);
  assert P(x);
  …
}
```

The need for termination:

functions

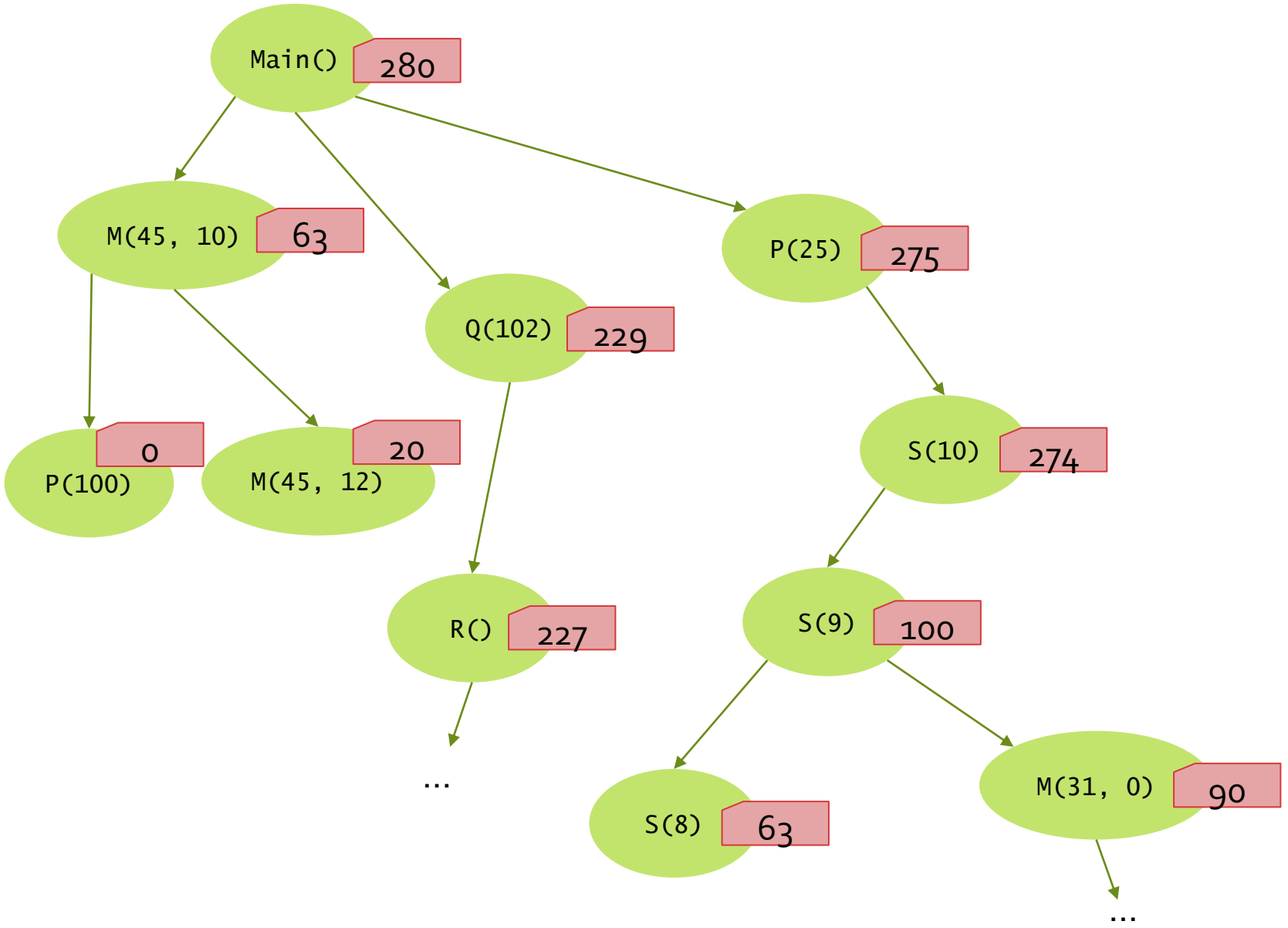
```
function F(x: int): int {
  1 + F(x)
}

method Caller()
{
  assert F(5) == 1 + F(5);
  assert false;
  …
}
```

How to establish termination



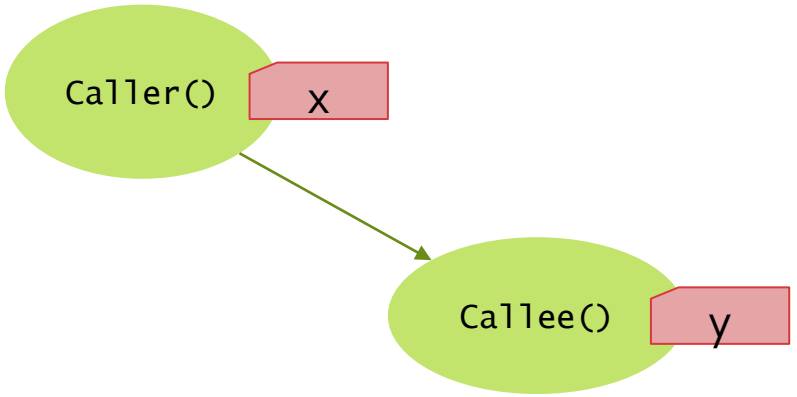
How to establish termination:

associate a *termination metric* with each method activation



Desired property for each call

If each call



satisfies $x \succ y$

in a fixed well-founded order $\succ$,

then all calls in the program terminate

Well-founded order

An irreflexive partial order $\succ$ on a set $\mathcal{A}$ is

well-founded

if there is no infinite descending chain

$$a_0 \succ a_1 \succ a_2 \succ a_3 \succ \ldots$$

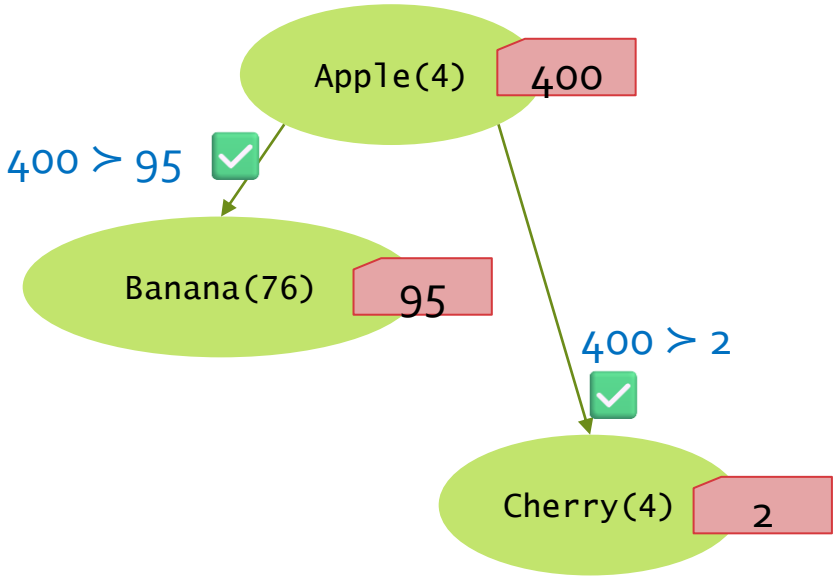
Example: The natural numbers with the usual greater-than ordering > is well-founded.

Defining termination metrics

- A `decreases` clause defines the termination metric for each method/function activation
- It is evaluated *on entry* to the method/function

```
method Apple(m: nat)
  decreases 100 * m
{
  if 1 <= m < 50 {
    Banana(80 - m);
    var c := Cherry(m);
  }
}

method Banana(n: nat)
  decreases n + 19
…

function Cherry(p: nat): nat
  decreases p / 2
…
```

For an activation `Apple(4)`

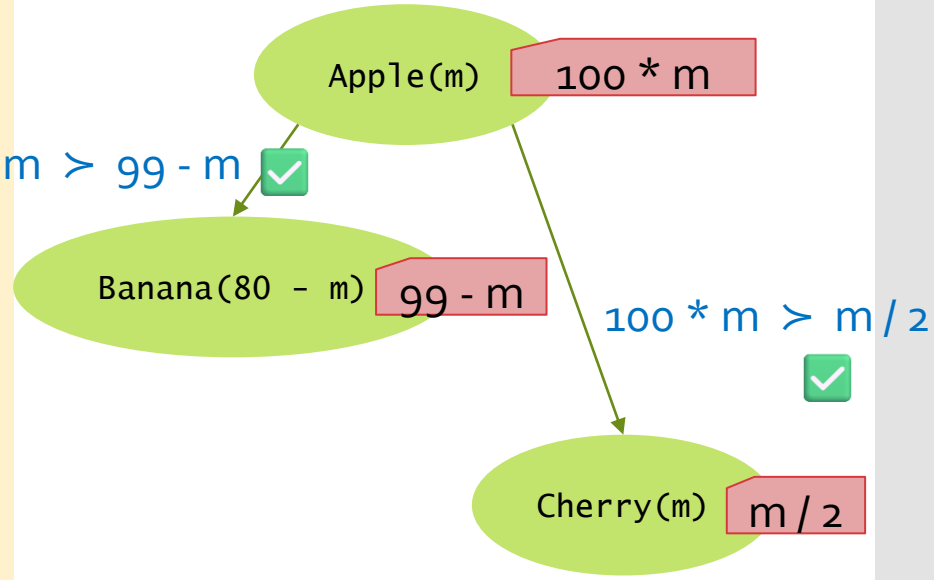


Defining termination metrics

- A `decreases` clause defines the termination metric for each method/function activation
- It is evaluated *on entry* to the method/function

```
method Apple(m: nat)
  decreases 100 * m
{
  if 1 <= m < 50 {
    Banana(80 - m);
    var c := Cherry(m);
  }
}

method Banana(n: nat)
  decreases n + 19
…

function Cherry(p: nat): nat
  decreases p / 2
…
```

For a general activation of `Apple`



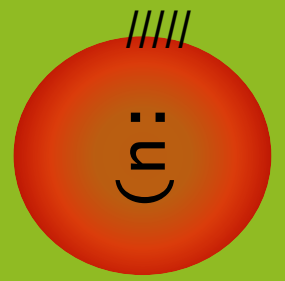
Examples

```
function Triangle(n: nat): nat
  decreases n
{
  if n == 0 then 0 else n + Triangle(n - 1)
}

function Factorial(n: nat): nat
  decreases n
{
  if n > 0 then n * Factorial(n - 1) else 1
}

function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

Default decreases clause

```
function Triangle(n: nat): nat
  decreases n
{
  if n == 0 then 0 else n + Tr
}

function Factorial(n: nat): na
  decreases n
{
  if n > 0 then n * Factorial(
}

function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n
}
```

An omitted `decreases` clause defaults to the method/function parameter



SumRange.dfy

More examples

Manual `decreases` clause

```
function SumRange(s: seq<int>, lo: int, hi: int): int
  requires 0 <= lo <= hi <= |s|
  decreases hi - lo
{
  if hi == lo then 0 else s[lo] + SumRange(s, lo+1, hi)
}
```

Pow.dfy

More examples

More interesting proof

```
// Return 2^n
opaque function Pow(n: nat): nat
{
  if n == 0 then 1 else 2 * Pow(n - 1)
}

lemma PowDistributesOverAddition(m: nat, n: nat)
  ensures Pow(m + n) == Pow(m) * Pow(n)
{
  if m == 0 {
    reveal Pow(); // def. Pow
  } else {
    calc {
      Pow(m + n);
    ==  { reveal Pow(); } // def. Pow
      2 * Pow(m - 1 + n);
    ==  { PowDistributesOverAddition(m - 1, n); } // I.H.
      2 * (Pow(m - 1) * Pow(n));
    ==
      2 * Pow(m - 1) * Pow(n);
    ==  { reveal Pow(); } // def. Pow
      Pow(m) * Pow(n);
    }
  }
}
```

TwoSort.dfy

Sort with boolean keys

```
type Payload
datatype Data = Data(key: bool, payload: Payload)

method TwoSort(a: array<Data>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> !a[i].key || a[j].key
```

TwoSort.dfy

Sort with boolean keys: solution (variation 0)

```
method TwoSort(a: array<Data>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> !a[i].key || a[j].key
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 <= lo <= hi <= a.Length
    invariant forall i :: 0 <= i < lo ==> !a[i].key
    invariant forall i :: hi <= i < a.Length ==> a[i].key
    decreases hi - lo
  {
    if !a[lo].key {
      lo := lo + 1;
    } else if a[hi - 1].key {
      hi := hi - 1;
    } else {
      a[lo], a[hi - 1] := a[hi - 1], a[lo];
      lo := lo + 1;
      hi := hi - 1;
    }
  }
}
```

TwoSort.dfy

Sort with boolean keys: solution (variation 1)

```
method TwoSort(a: array<Data>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> !a[i].key || a[j].key
{
  var lo, hi := 0, a.Length;
  while lo != hi
    invariant 0 <= lo <= hi <= a.Length
    invariant forall i :: 0 <= i < lo ==> !a[i].key
    invariant forall i :: hi <= i < a.Length ==> a[i].key
    decreases hi - lo, lo < a.Length && a[lo].key
  {
    if !a[lo].key {
      lo := lo + 1;
    } else if a[hi - 1].key {
      hi := hi - 1;
    } else {
      a[lo], a[hi - 1] := a[hi - 1], a[lo];
    }
  }
}
```

TwoSort.dfy

Sort with boolean keys: solution (variation 2)

```
method TwoSort(a: array<Data>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> !a[i].key || a[j].key
{
  if a.Length <= 1 {
    return;
  }
  var lo, hi := 0, a.Length;
  while lo + 1 != hi
    invariant 0 <= lo < hi <= a.Length
    invariant forall i :: 0 <= i < lo ==> !a[i].key
    invariant forall i :: hi <= i < a.Length ==> a[i].key
    decreases hi - lo, a[lo].key
  {
    if !a[lo].key {
      lo := lo + 1;
    } else if a[hi - 1].key {
      hi := hi - 1;
    } else {
      a[lo], a[hi - 1] := a[hi - 1], a[lo];
    }
  }
}
```

Pow.dfy

Strong induction

FastPow

```
opaque function FastPow(n: nat): nat {
  if n == 0 then
    1
  else
    var p := FastPow(n / 2);
    if n % 2 == 0 then
      p * p
    else
      2 * p * p
}

lemma Same(n: nat)
  ensures FastPow(n) == Pow(n)
```

Pow.dfy

FastPow: solution

```
lemma Same(n: nat)
  ensures FastPow(n) == Pow(n)
{
  if n == 0 {
    reveal Pow(), FastPow();
  } else {
    var p := FastPow(n / 2);
    var q := Pow(n / 2);
    calc {
      FastPow(n);
    ==  { reveal FastPow(); }
      if n % 2 == 0 then p * p else 2 * p * p;
    ==  { Same(n / 2); }
      if n % 2 == 0 then q * q else 2 * q * q;
    ==  { PowDistributesOverAddition(n / 2, n / 2); }
      if n % 2 == 0 then Pow(n) else 2 * Pow(n - 1);
    ==  { reveal Pow(); }
      if n % 2 == 0 then Pow(n) else Pow(n);
    ==
      Pow(n);
    }
  }
}
```

Well-founded order for datatypes

Inductive datatypes are ordered by *structural inclusion*

Example:

```
datatype List<X> = Nil | Cons(T, List<X>)
```

- `Cons(head, tail)` $\succ$ `head` (*)
- `Cons(head, tail)` $\succ$ `tail`

(*) provided **head** is a datatype value

Well-founded ordered for other types

type	X ≻ y
inductive datatype	X structurally includes y
`int`	
`real`	
`bool`	
`object?`	
`set<T>`	
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \quad \wedge \quad y < X$
`real`	
`bool`	
`object?`	
`set<T>`	
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	
`bool`	
`object?`	
`set<T>`	
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	$0.0 \leq X \ \wedge \ y + 1.0 \leq X$
`bool`	
`object?`	
`set<T>`	
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	X ≻ y
inductive datatype	X structurally includes y

`int`	`0 ≤ X ∧ y < X`
	equivalently:
	`0 ≤ X ∧ y + 1 ≤ X`

`real`	`0.0 ≤ X ∧ y + 1.0 ≤ X`
`bool`	`X ∧ ¬y`
`object?`	
`set<T>`	
`seq<T>`	
`iset<T>`	

Example: `Xy.dfy`

Xy.dfy

Example

```
// In Lustre:
//    x  =  if c then y else 0
//    y  =  if c then 1 else x

function x(c: bool): int
  decreases c
{
  if c then y(c) else 0
}

function y(c: bool): int
  decreases !c
{
  if c then 1 else x(c)
}
```

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	$0.0 \leq X \ \wedge \ y + 1.0 \leq X$
`bool`	$X \wedge \neg y$
`object?`	$X \neq \texttt{null} \ \wedge \ y = \texttt{null}$
`set<T>`	
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	$0.0 \leq X \ \wedge \ y + 1.0 \leq X$
`bool`	$X \wedge \neg y$
`object?`	$X \neq \texttt{null} \ \wedge \ y = \texttt{null}$
`set<T>`	$y \subsetneq X$
`seq<T>`	
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	$0.0 \leq X \ \wedge \ y + 1.0 \leq X$
`bool`	$X \wedge \neg y$
`object?`	$X \neq$ `null` $\wedge \ y =$ `null`
`set<T>`	$y \subsetneq X$
`seq<T>`	y is proper subsequence of X
`iset<T>`	

Well-founded ordered for other types

type	$X \succ y$
inductive datatype	X structurally includes y
`int`	$0 \leq X \ \wedge \ y < X$ equivalently: $0 \leq X \ \wedge \ y + 1 \leq X$
`real`	$0.0 \leq X \ \wedge \ y + 1.0 \leq X$
`bool`	$X \wedge \neg y$
`object?`	$X \neq$ `null` $\wedge \ y =$ `null`
`set<T>`	$y \subsetneq X$
`seq<T>`	y is proper subsequence of X
`iset<T>`	`false`

Union of types

Let $(\mathcal{A}, \succ_{\mathcal{A}})$ and $(\mathcal{B}, \succ_{\mathcal{B}})$ be well-founded orders, where $\mathcal{A}$ and $\mathcal{B}$ are disjoint.

Define $\succ$ on $\mathcal{A} \cup \mathcal{B}$ by

$$x \succ y \quad = \quad x \succ_{\mathcal{A}} y \;\; \vee \;\; x \succ_{\mathcal{B}} y$$

Then, $(\mathcal{A} \cup \mathcal{B}, \succ)$ is also a well-founded order.

Top element

Let $(\mathcal{A}, \succ)$ be a well-founded order.

Let $\top$ denote an element not in $\mathcal{A}$.

Define $\mathcal{A}_\top$ to be $\mathcal{A} \cup \{\top\}$.

Define $\succ_\top$ on $\mathcal{A}_\top$ by

$$x \succ_\top y \quad = \quad x \succ y \ \vee \ (x = \top \ \wedge \ y \neq \top)$$

Then, $(\mathcal{A}_\top, \succ_\top)$ is also a well-founded order.

Lexicographic orders

Let $(\mathcal{A}, \succ_{\mathcal{A}})$, $(\mathcal{B}, \succ_{\mathcal{B}})$, and $(\mathcal{C}, \succ_{\mathcal{C}})$ be well-founded orders.

Define $\succ$ on the (so-called *lexicographic*) tuples $\mathcal{A} \times \mathcal{B} \times \mathcal{C}$ by

$$
\begin{aligned}
& a_0, b_0, c_0 \succ a_1, b_1, c_1 \\
& = \\
& \quad a_0 \succ_{\mathcal{A}} a_1 \vee \\
& \quad (a_0 = a_1 \wedge b_0 \succ_{\mathcal{B}} b_1) \vee \\
& \quad (a_0 = a_1 \wedge b_0 = b_1 \wedge c_0 \succ_{\mathcal{C}} c_1)
\end{aligned}
$$

Then, $(\mathcal{A} \times \mathcal{B} \times \mathcal{C}, \succ)$ is a well-founded order.

Lexicographic tuples of different lengths

- How to compare elements of $\mathbb{Z}^5$ and $\mathbb{Z}^7$?
- Examples:
  - $9, 2, 2, 7, 1 \overset{?}{\succ} 8, 6, 7, 5, 3, 0, 9$ (a)
  - $8, 6, 7, 5, 3 \overset{?}{\succ} 8, 6, 7, 5, 3, 0, 9$ (b)
- Some candidates:
  - Shorter always smaller
  - For equal prefixes, shorter is smaller
  - For equal prefixes, shorter is larger (*)
  - Pad with T and then compare the equal lengths (*)

(*) Provided there's an upper bound on the length

Examples

- Pad with T and then compare the equal lengths
  - 9, 2, 2, 7, 1 $\succ$ 8, 6, 7, 5, 3, 0, 9 (true)
    =
    9, 2, 2, 7, 1, T, T $\succ$ 8, 6, 7, 5, 3, 0, 9
  - 8, 6, 7, 5, 3 $\succ$ 8, 6, 7, 5, 3, 0, 9 (true)
    =
    8, 6, 7, 5, 3, T, T $\succ$ 8, 6, 7, 5, 3, 0, 9
  - 8, 6, 7, 5, 3, 0, 9 $\succ$ 9, 2, 2, 7, 1 (false)
    =
    8, 6, 7, 5, 3, 0, 9 $\succ$ 9, 2, 2, 7, 1, T, T
- Note that additional T-padding does not change anything
  - a, b, c, d, e $\succ$ v, w, x, y, z
    =
    a, b, c, d, e, T $\succ$ v, w, x, y, z, T

Termination metrics in Dafny

- Dafny fixes a well-founded order for each type
- Its termination metrics use (*)

$$\underbrace{\mathcal{A}_\mathsf{T} \times \mathcal{A}_\mathsf{T} \times \cdots \times \mathcal{A}_\mathsf{T}}_{k}$$

  where $\mathcal{A}$ is the union of all types and
  k is the length of the longest given `decreases` clause in the program
- `decreases` clauses are automatically T-padded to the longest length

(*) Almost. One addition added in later lecture.

Example: Ackermann

```
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))
}
```

m, n ≻ m-1, 1 ✅

m, n ≻ m, n-1 ✅

m, n ≻ m-1, … ✅

Example

```
function P(x: int): bool
  //...

// Count the number of values less than "n" that satisfy "P"
function CountPs(n: nat): int
{
  if n == 0 then
    0
  else if P(n - 1) then
    1 + CountPs(n - 1)
  else
    CountPs(n - 1)
}
```

Drop

```
datatype List<X> = Nil | Cons(head: X, tail: List<X>)

function Length(xs: List): nat {
  if xs == Nil then 0 else 1 + Length(xs.tail)
}

function Drop(xs: List, n: nat): List
  requires n <= Length(xs)
{
  if n == 0 then xs else Drop(xs.tail, n - 1)
}

function Drop'(xs: List, n: nat): (r: List)
  requires n <= Length(xs)
  ensures Length(r) == Length(xs) - n
{
  if n == 0 then xs else Drop'(xs, n - 1).tail
}
```

```
lemma {:induction false} Same(xs: List, n: nat)
  requires n <= Length(xs)
  ensures Drop(xs, n) == Drop'(xs, n)
```

Notice the 3 different uses of the inductive hypothesis

Drop: solution

```
lemma {:induction false} Same(xs: List, n: nat)
  requires n <= Length(xs)
  ensures Drop(xs, n) == Drop'(xs, n)
{
  if n < 2 {
    // easy
  } else {
    calc {
      Drop(xs, n);
    ==  // def. Drop
      Drop(xs.tail, n - 1);
    ==  { Same(xs.tail, n - 1); } // I.H.
      Drop'(xs.tail, n - 1);
    ==  // def. Drop'
      Drop'(xs.tail, n - 2).tail;
    ==  { Same(xs.tail, n - 2); } // I.H.
      Drop(xs.tail, n - 2).tail;
    ==  // def. Drop
      Drop(xs, n - 1).tail;
    ==  { Same(xs, n - 1); } // I.H.
      Drop'(xs, n - 1).tail;
    ==  // def. Drop'
      Drop'(xs, n);
    }
  }
}
```

Layering methods/functions

Given

```
function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

Specify termination of

```
function SumOfFibs(xs: List<nat>): nat
  decreases ??
{
  if xs == Nil then 0 else
    Fib(xs.head) + SumOfFibs(xs.tail)
}
```

Layering methods/functions

One possibility:

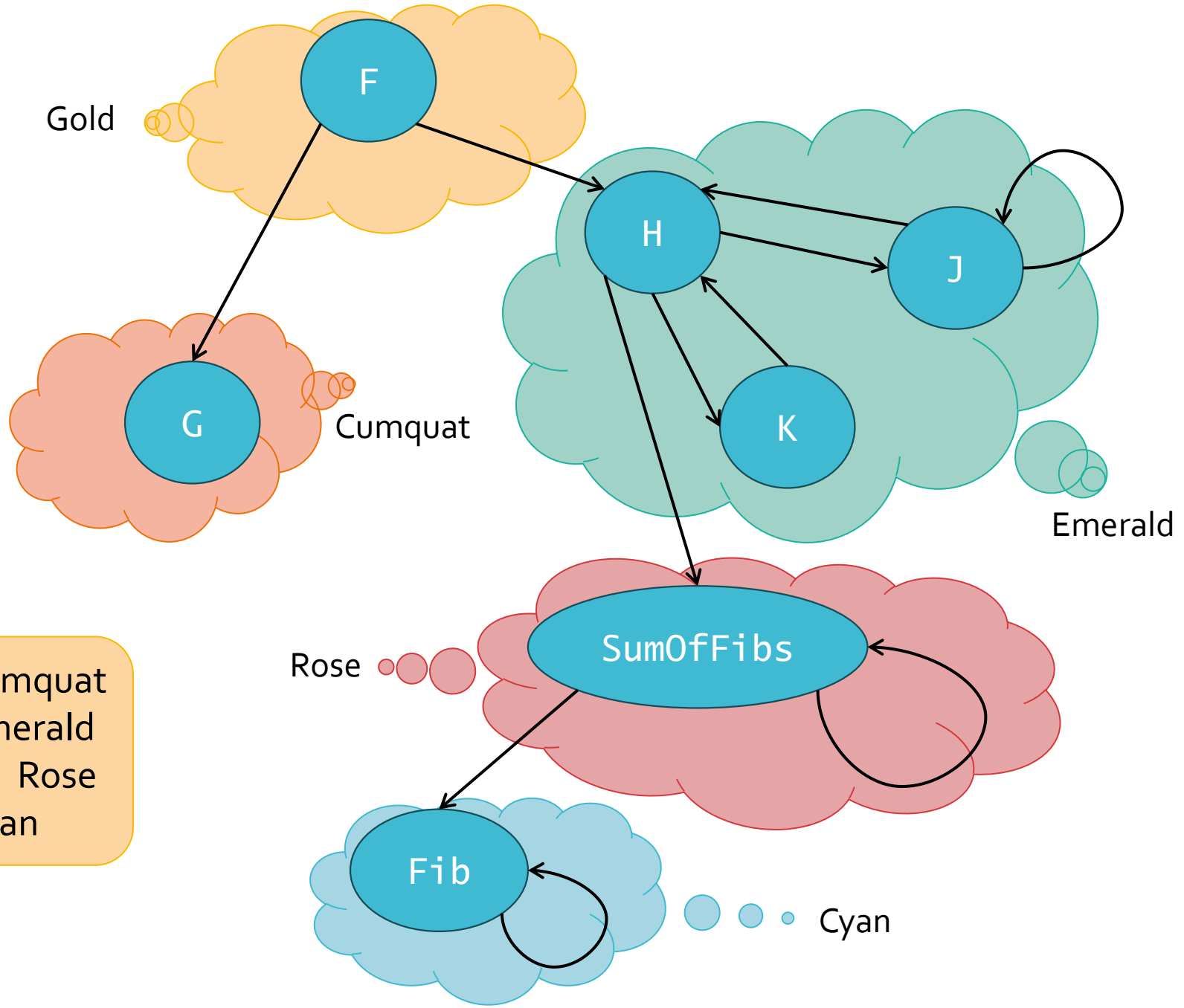
```
function Fib(n: nat): nat
  decreases 0, n
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

```
function SumOfFibs(xs: List<nat>): nat
  decreases 1, xs
{
  if xs == Nil then 0 else
    Fib(xs.head) + SumOfFibs(xs.tail)
}
```

But this requires *changing* the given specification of `Fib` 😧

Call graph

Gold ≻ Cumquat
Gold ≻ Emerald
Emerald ≻ Rose
Rose ≻ Cyan



Termination metrics in Dafny (updated)

- Dafny fixes a well-founded order for each type
- Its termination metrics use

$$\mathcal{C} \times \underbrace{\mathcal{A}_\top \times \mathcal{A}_\top \times \cdots \times \mathcal{A}_\top}_{k}$$

where

  - $\mathcal{A}$ is the union of all types
  - $\mathcal{C}$ is the set of strongest connected components (SCCs) of the call graph
  - k is the length of the longest given `decreases` clause in the program

- `decreases` clauses are automatically padded to the longest length

Layering methods/functions

Solution

```
function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

```
function SumOfFibs(xs: List<nat>): nat
  decreases xs
{
  if xs == Nil then 0 else
    Fib(xs.head) + SumOfFibs(xs.tail)
}
```

The recurrence equation

$$AllPositive = \mathcal{F}(AllPositive)$$

has many solutions in $AllPositive$

We want the *greatest* solution

With $P = \mathcal{F}(P)$

What to prove	How to prove it
$A \implies \mu\mathcal{F}$	
$\mu\mathcal{F} \implies B$	
$A \implies \nu\mathcal{F}$	
$\nu\mathcal{F} \implies B$	

With $P = \mathcal{F}(P)$

What to prove	How to prove it
$A \implies \mu\mathcal{F}$	$A \implies P$
$\mu\mathcal{F} \implies B$	
$A \implies \nu\mathcal{F}$	
$\nu\mathcal{F} \implies B$	$P \implies B$

With $P = \mathcal{F}(P)$

What to prove	How to prove it
$A \implies \mu\mathcal{F}$	$A \implies P$
$\mu\mathcal{F} \implies B$	via iterates
$A \implies \nu\mathcal{F}$	via iterates
$\nu\mathcal{F} \implies B$	$P \implies B$

Iterates

Least solution

- $P = \mathcal{F}(P)$
- $P^{\#}{}_k = \mathcal{F}^k(\bot)$
- $P = \bigvee_k P^{\#}{}_k$

Greatest solution

- $P = \mathcal{F}(P)$
- $P^{\#}{}_k = \mathcal{F}^k(\top)$
- $P = \bigwedge_k P^{\#}{}_k$

provided $\mathcal{F}$ is *continuous*

Continuity

- $\mathcal{F}$ is continuous
  =
  $\mathcal{F}$ distributes over arbitrary joins (for least fixpoints) / meets (for greatest fixpoints)

- For least fixpoints, we cannot allow:

$$P = \ldots \forall x \cdot \ldots P \ldots$$

- For greatest fixpoints, we cannot allow:

$$P = \ldots \exists x \cdot \ldots P \ldots$$

Iterates

For least fixpoints: (and dually for greatest fixpoints)

- $P^{\#}_{0} = \bot$
- $P^{\#}_{i+1} = \mathcal{F}(P^{\#}_{i})$
- $P^{\#}_{\lambda} = \bigvee_{o<\lambda} \mathcal{F}^{o}$

- $P = \bigvee_{o} P^{\#}_{o}$

regardless of whether $\mathcal{F}$ is *continuous*

With $P = \mathcal{F}(P)$

What to prove	How to prove it
$A \implies \mu\mathcal{F}$	$A \implies P$
$\mu\mathcal{F} \implies B$	`least lemma`
$A \implies \nu\mathcal{F}$	`greatest lemma`
$\nu\mathcal{F} \implies B$	$P \implies B$

Summary

- You can call whatever you want whenever you want, provided
  - you satisfy the precondition, and
  - you can show a decrease in some fixed well-founded order
- Examples:
  - `FastPow`: strong induction
  - `SubstIdempotent`: mutual recursion/induction
  - `x/y`: custom ordering
  - `MultComm`: ad-hoc cases
  - Call chains: use constants, call-graph SCC
- Extreme predicates
  - defined as least or greatest fix-points
  - recursive calls have to be in positive positions, but need not terminate
  - reasoning uses analogous extreme lemmas

Program safely!