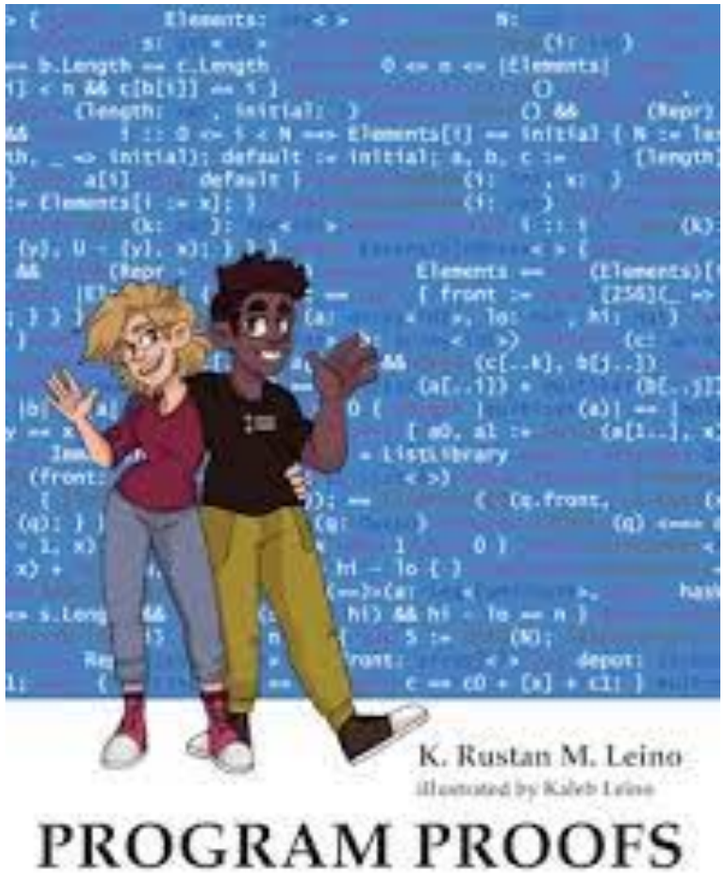
CS 456

Programming Languages
Fall 2024

Week 13
Dafny

Dafny

- Solver-aided language and verifier
- Language is statically-typed
- Imperative (with lots of functional language features)
- Compiles to C#, Java, Go, Python, …



Reference manual:

`https://dafny.org/dafny/DafnyRef/DafnyRef.html`

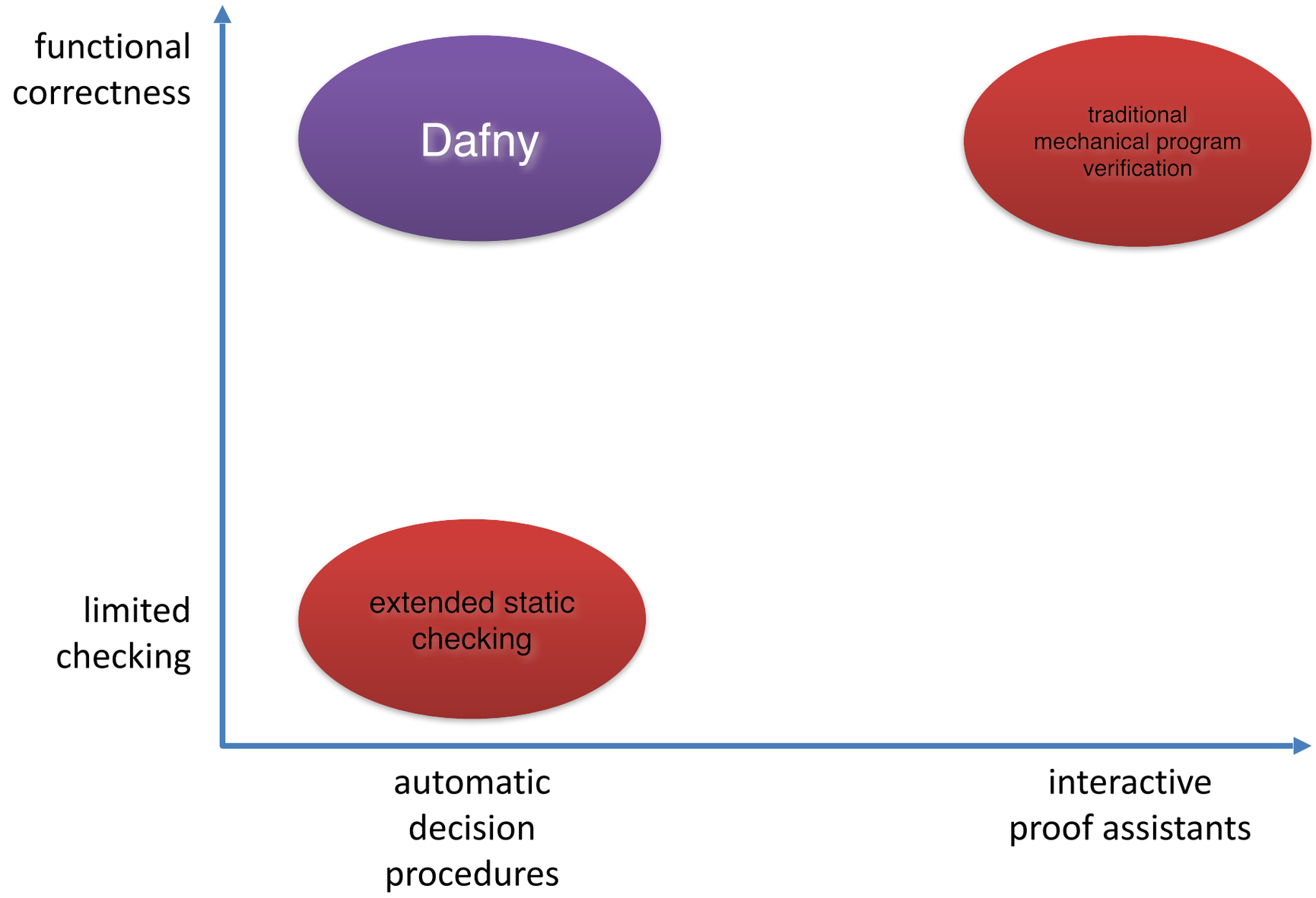
Dafny

- Applies Hoare reasoning to programs
- User provides specifications in the form of pre- and postconditions, along with other assertions
- Dafny verifies that the program meets the specification
  - When successful, Dafny guarantees (total) functional correctness of the program

Correctness:

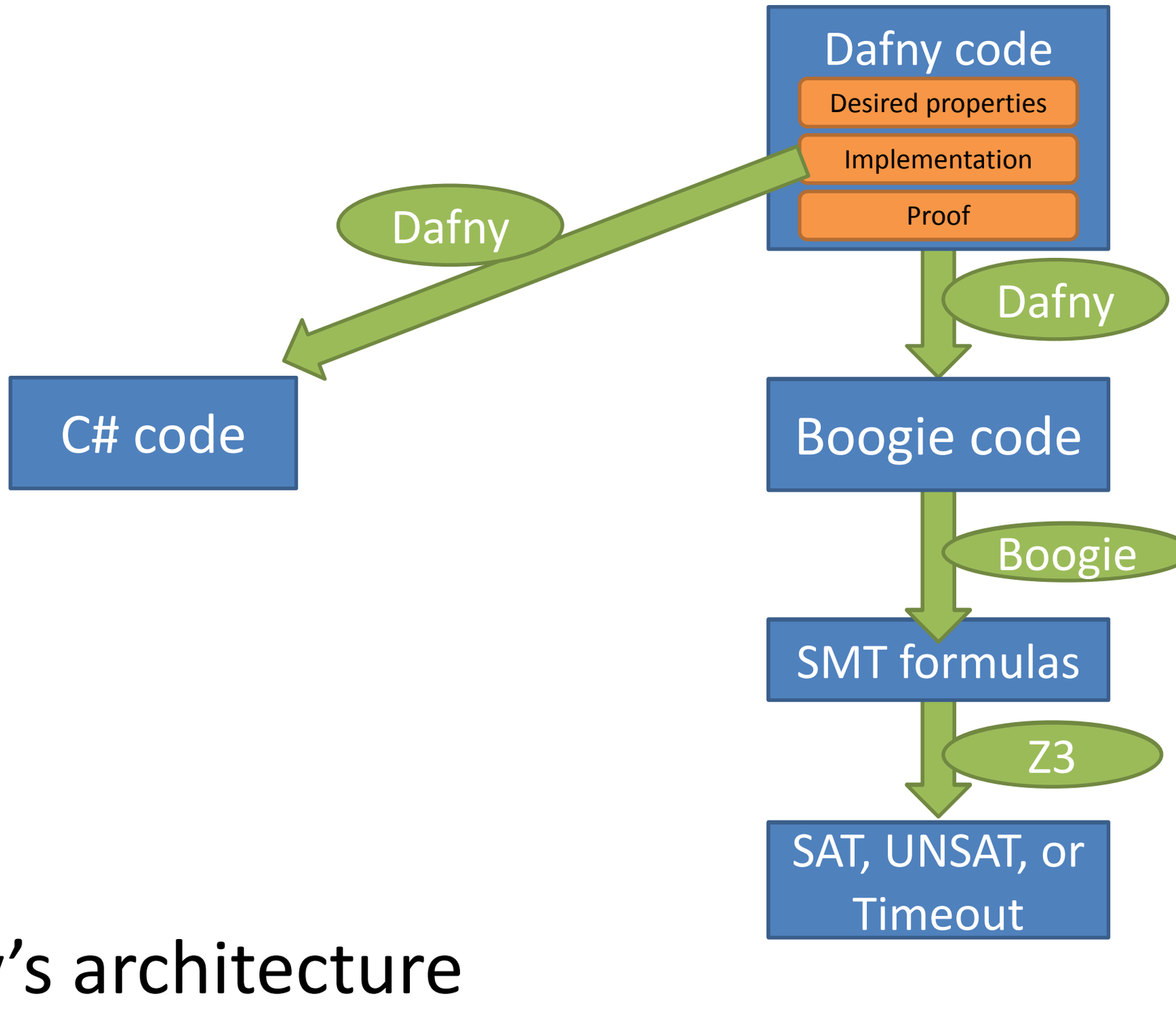
- Reflects base-level semantic properties (no runtime errors (e.g., divide-by-zero, null pointer dereferences, etc.)
- But, also justifies higher-level application-specific properties (e.g., correctness of distributed systems, …)

Types of Program Verification



Adapted from slides of Bryan Parno

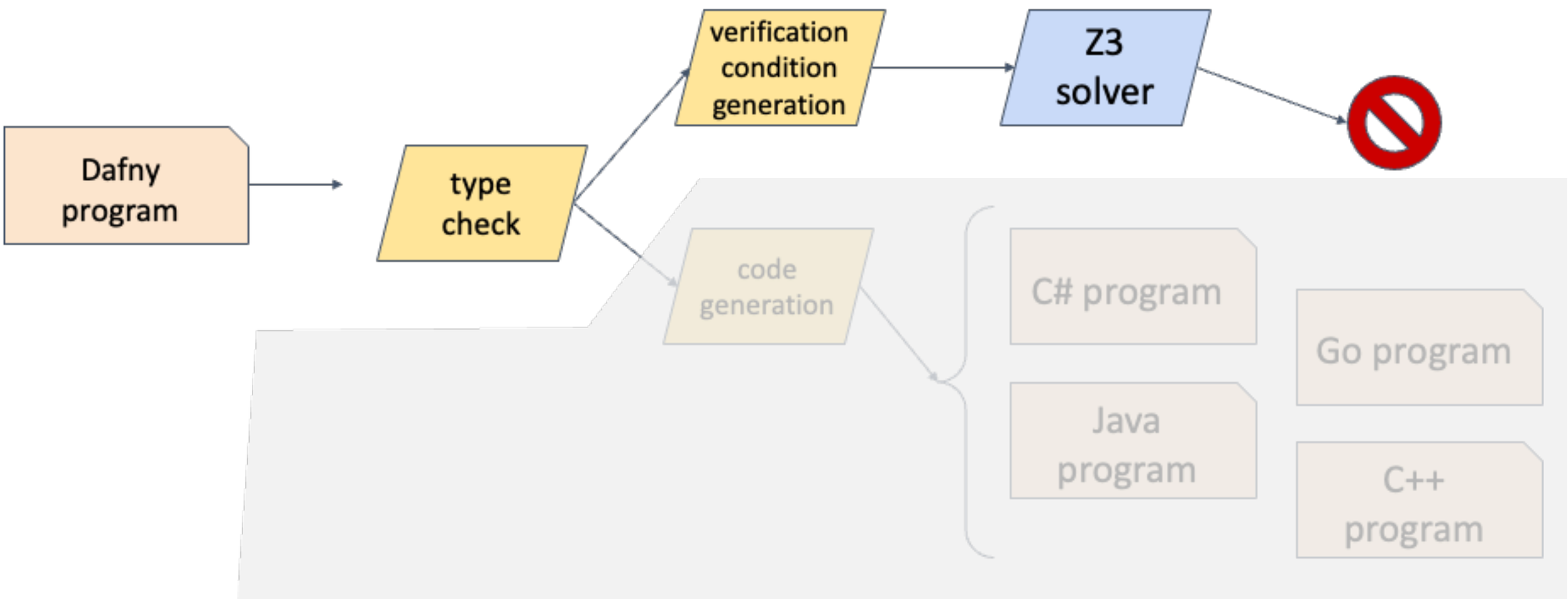
Architecture



Dafny's architecture

Adapted from slides of Bryan Parno

Pipeline



Adapted from slides of Jon Howell and Manos Kapritsos

Specifications

- Specifications are meant to capture salient behavior of an application, eliding issues of efficiency and low-level representation.

```
forall k:int :: 0 <= k < a.Length ==> 0 < a[k]
```

- Specifications in Dafny can be arbitrarily sophisticated.

- We can think of Dafny as being two smaller languages rolled into one:

  - An imperative core that has methods, loops, arrays, if statements... and other features found in realistic programming languages. This core can be compiled and executed.

  - A pure (functional) specification language that supports functions, sets, predicates, algebraic datatypes, etc. This language is used by the prover but is not compiled.

Examples


```
method Triple (x: int) returns (r : int) {
  var y := 2 * x;
  r := x + y;
  assert r == 3 * x;
}

method Caller () {
  var t := Triple(9);
  assert t == 27;      // assert fails: why?
}

method TripleSpec (x: int) returns (r : int)
   ensures r == 3 * x
{
      var y := 2 * x;
      r := x + y;
}

method CallerSpec () {
  var t := TripleSpec(9);
   assert t == 27; // assert succeeds
}
```

Examples

// Valid method

```
method Index (n: int) returns (i : int)
    requires 1 <= n
    ensures 0 <= i <= n
{
    i := n /2;
}
```

// Invalid assert - how would you fix this?

```
method CallIndex() {
    var t1 := Index(50);
    var t2 := Index(50);
    assert t1 == t2;
}
```

Examples


```
method Min (x : int, y : int) returns (m : int)
  ensures m <=x && m <= y
{
   m := if x <= y then x else y;
}
```

The implementation satisfies the spec but does not capture the intended behavior!

```
method Min (x : int, y : int) returns (m : int)
  ensures m <=x && m <= y
  ensures m == x || m == y
{
   m := if x <= y then x else y;
}
```

Functions vs. Methods

- Functions in Dafny have no computational effect
  - Deterministic
  - Can be used in specifications!

```
function average(a: int, b: int): int {
  (a + b) / 2
}

method Triple (x: int) returns (r: int)
   ensures average(r, 3 * x) == 3 * x
{
  if (x < 0) { return -x; } else { return x; }
}
```

Alternative definition:

```
function average(a: int, b: int): int
  requires 0 <= a && 0 <= b
{
  (a + b) / 2
}
```

Functions


```
function fib(n: nat): nat
{
   if n == 0 then 0 else
   if n == 1 then 1 else
             fib(n - 1) + fib(n - 2)
}

method Fib (n: nat) returns (x: nat)
   ensures x == fib(n);
{
  var i := 0;
  x := 0;
  var y := 1;
  while (i < n) {
    x, y := y, x+y;
    i := i + 1;
   }
}
```

Dafny fails to verify this program. Why?

Invariants

- Follows the same principle as Hoare logic

```
method ComputeFib (n: nat) returns (y: nat)
    ensures y == fib(n);
{
    if (n == 0) { return 0; }

  var i := 1;
  var x := 0;
  y := 1;

   while (i < n)
      invariant 0 < i <= n
      invariant x == fib (i - 1)
      invariant y == fib (i)
   {
      x, y := y, x+y;
      i := i + 1;
   }
}
```

Invariants


```
method loopEx (n : nat)
{
    var i : int := 0;
    while (i < n)
        invariant 0 <= i
        {
            i := i + 1;
    }
    assert i == n;
}
```

Dafny will not verify this program. Why?

Need invariants to be inductive!

- hold in the initial state
- hold in every state reachable from the initial state
- strong enough to imply the postcondition

```
method loopExCheckFixed (n : nat)
{
    var i : int := 0;
    while (i < n)
        invariant 0 <= i <= n
        {
            i := i + 1;
    }
    assert i == n;
}
```

Ghost vs. Compiled

- Ghost constructs are syntactic forms used only in specifications
  - Pre- (requires) and post- (ensures) conditions are ghost constructs
  - As are assert and invariant

  - Some constructs such as functions exist in both ghost and compiled form
  - Can explicitly declare variables, parameters, methods, etc. as ghost; such objects are not compiled into executables
    - ★ Cannot assign ghost entities to complied ones

```
method Triple(x : int) returns (r: int)
  ensures r = 3 * x
{
   var y := 2 * x;
   r := x + y;
   ghost var a, b := DoubleQuadruple(x);
   assert a <= r <= b || b <= r <= a;
}

ghost method DoubleQuadruple (x : int) returns (a: int, b: int)
   ensures a = 2 * x && b = 4 * x
{
   a := 2 * x;
   b := 2 * a;
}
```

Assert

- `assert E` is a no-op if `E` holds, otherwise program faults.
- To show postcondition `Q` holds, i.e.,

  `WP(assert E, Q)`

  we must prove `E && Q`
- Backward reasoning

Alternative interpretation:

- `assert E` evaluates `E` and if `E` does not crash, continues
- No proof obligations introduced
- Forward reasoning

Concept Check


```
method MultipleReturnsSpec(x: int, y: int) returns (more: int, less: int)
{
  more := x + y;
  less := x - y;
}
```

What is a meaningful spec for this method?

```
method MultipleReturnsSpec(x: int, y: int) returns (more: int, less: int)
  ensures less < x
  ensures x < more
{
  more := x + y;
  less := x - y;
}
```

Concept Check


```
method Max (a : int, b : int) returns (c : int)
{
    if (a < b) {
        c := b;
    }
    else { c := a; }
}
```

What is a meaningful spec for this method?

```
method Max (a : int, b : int) returns (c : int)
    ensures (a <= c && b <= c) && (b == c || a == c)
{
    if (a < b) {
        c := b;
    }
    else { c := a; }
}
```

Concept Check


```
method Abs(x: int) returns (r: int)
  ensures r >= 0
   {
    if (x < 0)
      { return -x; }
    else
      { return x; }
  }
```

What's wrong with this spec? How would you fix it?

```
method AbsFixed(x: int)    returns (y: int)
  ensures 0 <= x ==> y == x
  ensures x < 0 ==> y == -x
{
  if (x < 0) { return -x; }
     else { return x; }
}
```

```
method AbsFixedA(x: int)   returns (y: int)
    ensures 0 <= y && ( y == x || y == -x)
{
  if (x < 0) { return -x; }
     else { return x; }
}
```

Basic setup

- Specify correctness conditions as pre/post-conditions that can be checked (mostly) automatically using a WP inference procedure
- But, not all properties we wish to verify can be expressed in terms of actions on the transition relation defined by axiomatic rules

Need proof techniques that allow us to verify properties over:

1. Inductive datatypes (e.g., lists, trees, …)
2. Semantic objects (e.g., heaps)
3. Imperative data structures (e.g, arrays)

Additionally, Dafny verifies total correctness

- Hoare rules only assert partial correctness properties
- Need additional insight to reason about termination

Decreases clause


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

Dafny complains that it cannot prove the recursive call terminates - it is unable to identify a termination metric that signals every recursive call gets "smaller"

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

What about using -lo as a decreases clause?

Examples


```
function F(x : int) : int {
  if x < 10 then x else F(x - 1)

}
```

Are decreases clauses required?

```
function G(x : int) : int {
   if 0 <= x then G(x - 2) else x
}
```

```
function H(x : int) : int
  decreases x + 60
{
   if x < -60 then x else H (x - 1)
}
```

Are decreases clauses required here?

Why would `decreases x` not work?

```
function L(x: int) : int
  decreases 100 - x
{
   if x < 100 then L(x + 1) + 10 else x
}
```

Are decreases clauses required here?

Well-Founded Relations

A binary relation $\succeq$ is well-founded if it is:

- irreflexive
- transitive
- satisfies a descending chain condition, i.e. there is no infinite sequence of values a0, a1, … such that a0 $\succeq$ a1 $\succeq$ …

Can establish such a relation for any datatype

- E.g., Booleans (true $\succeq$ false), a less-than ordering relation on integers, or a subset relation on a set

```
function M(x: int, b: bool) : int
  decreases if b then 0 else 1
{
  if b then x else M(x + 25, true)
}
```

Lexiographic Tuples

- Component-wise comparison of decreases clauses:

$4, 12 \succeq 4, 11$

$4, 6, 0 \succeq 4, 6, 0, 25, 3$

$2, 5 \succeq 1$

What decreases clause is necessary to allow Dafny to verify this program?

```
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))
}
```