

CS 565

Programming Languages (graduate)
Spring 2025

Week 11

Loop Invariants and Introduction
to Dafny

Loop Invariants

2

Hoare Logic is a structural model-theoretic proof system

- Rules characterize a set of states consistent with the requirements imposed by the pre- and post-conditions
- Highly mechanical: intermediate states can almost always be automatically constructed
- One major exception:

$$\frac{\vdash \{I \wedge b\} c \{I\}}{\vdash \{I\} \text{ while } b \text{ do } c \text{ end } \{I \wedge \neg b\}} \text{ HLWHILE}$$

The invariant must:

- be weak enough to be implied by the precondition
- hold across each iteration
- be strong enough to imply the postcondition

Example

3

```
{{ True }}  
  if X <= Y then  
    {{  
      Z := Y - X  
    }}  
  else  
    {{  
      Y := X + Z  
    }}  
  end  
{{ Y = X + Z }}
```

Our proof rules provide
a systematic way of
generating intermediate
assertions. The
fully decorated program
constitutes a proof that
the program when executed
in a state that satisfies the
precondition, will produce
a state satisfying the postcondition.

Example

4

```
{{ True }}  
  if X <= Y then  
    {{  
      Z := Y - X  
    }}  
  else  
    {{  
      Y := X + Z  
      {{ Y = X + Z }}  
    }}  
  end  
{{ Y = X + Z }}
```

follows from the postcondition

Example

5

```
{{ True }}  
  if X <= Y then  
    {{  
      Z := Y - X  
      {{Y = X + Z}}  
    else  
      {{  
        Y := X + Z  
        {{ Y = X + Z }}  
      end  
      {{ Y = X + Z }}
```

Example

6

```
{{ True }}  
  if X <= Y then  
    {{X <= Y}}  
    Z := Y - X  
    {{Y = X + Z}}  
  else  
    {{X > Y}}  
    Y := X + Z  
    {{Y = X + Z}}  
  end  
{{ Y = X + Z }}
```

- *By If Rule*
- *But, shape of precondition in assignments does not match the shape demanded by the Assgn rule*

Example

7

```
{{ True }}
if X <= Y then
    {{ X <= Y }} →
    {{ Y = X + (Y - X) }}
    Z := Y - X
    {{ Y = X + Z }}
else
    {{ X > Y }} →
    {{ X + Z = X + Z }}
    Y := X + Z
    {{ Y = X + Z }}
end
{{ Y = X + Z }}
```

*Strengthen precondition using
rule of consequence and Assgn
rule*

Non-trivial implication

Proof can be constructed automatically, reasoning
backwards from the postcondition

Loops

8

```
{{ True }} -->
    {{
X := a;
    {{
Y := b;
    {{
Z := 0;
    {{
while X <> 0 && Y <> 0 do
    {{
X := X - 1;
    {{
Y := Y - 1;
    {{
Z := Z + 1;
    {{
end
{{ Z = min a b }}
```

Loops

9

```
{ { True } }  
  { {  
    X := a;  
    { {  
      Y := b;  
      { {  
        Z := 0;  
        { {  
while X <> 0 && Y <> 0 do  
  { {  
    X := X - 1;  
    { {  
      Y := Y - 1;  
      { {  
        Z := Z + 1;  
        { {  
end  
{ { Z = min a b } }
```

postcondition given in terms
of inputs a and b

loop invariant expresses constraints
on local variables X and Y

candidate invariant:

$$Z + \min X Y = \min a b$$

Loops

10

```
{{ True }}
  {{ min a b = min a b }}
X := a;
  {{ min X b = min a b }}
Y := b;
  {{ min X Y = min a b }}
Z := 0;
  {{ Inv }}
while X <> 0 && Y <> 0 do
  {{ Z + 1 + min (X - 1) (Y - 1) = min a b }}
  X := X - 1;
  {{ Z + 1 + min X (Y - 1) = min a b }}
  Y := Y - 1;
  {{ Z + 1 + min X Y = min a b }}
  Z := Z + 1;
  {{ Inv }}
end
{{ ~(X <> 0 /\ Y <> 0) /\ Inv }} ->
{{ Z = min a b }}
```

$Inv == (Z + \min X Y = \min a b)$

Loops

11

```
{ { True } } ->
  { { min a b = min a b } }
```

```
X := a;
  { { min X b = min a b } }
Y := b;
  { { min X Y = min a b } }
Z := 0;
  { { Inv } }
```

```
while X <> 0 && Y <> 0 do
```

```
  { { Inv /\ (X <> 0) /\ Y <> 0 ) } } ->
  { { Z + 1 + min (X - 1) (Y - 1) = min a b } }
```

```
X := X - 1;
  { { Z + 1 + min X (Y - 1) = min a b } }
Y := Y - 1;
  { { Z + 1 + min X Y = min a b } }
Z := Z + 1;
  { { Inv } }
```

```
end
```

```
{ { ~(X <> 0 /\ Y <> 0) /\ Inv ) } } ->
{ { Z = min a b } }
```

This style of proof construction is known as weakest precondition inference

Identify a precondition that satisfies the largest set of states that still enable verification of the postcondition

Can automate this inference once we know the loop invariant

Concept Check

12

$\{\{ ? \}\}$ skip $\{\{ X = 5 \}\}$

$X = 5$

$\{\{ ? \}\}$ $X := Y + Z$ $\{\{ X = 5 \}\}$

$Y + Z = 5$

$\{\{ ? \}\}$ $X := Y$ $\{\{ X = Y \}\}$

True

$\{\{ ? \}\}$

if $X = 0$ then

$Y := Z + 1$

else $Y := W + 2$

$\{\{ Y = 5 \}\}$

$(X = 0 \wedge Z = 4) \vee (X \neq 0 \wedge W = 3)$

$\{\{ ? \}\}$ $X := 5$ $\{\{ X = 0 \}\}$

False

$\{\{ ? \}\}$ while true do

$X := 0$

end

$\{\{ X = 0 \}\}$

True

Dafny

13

- 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>

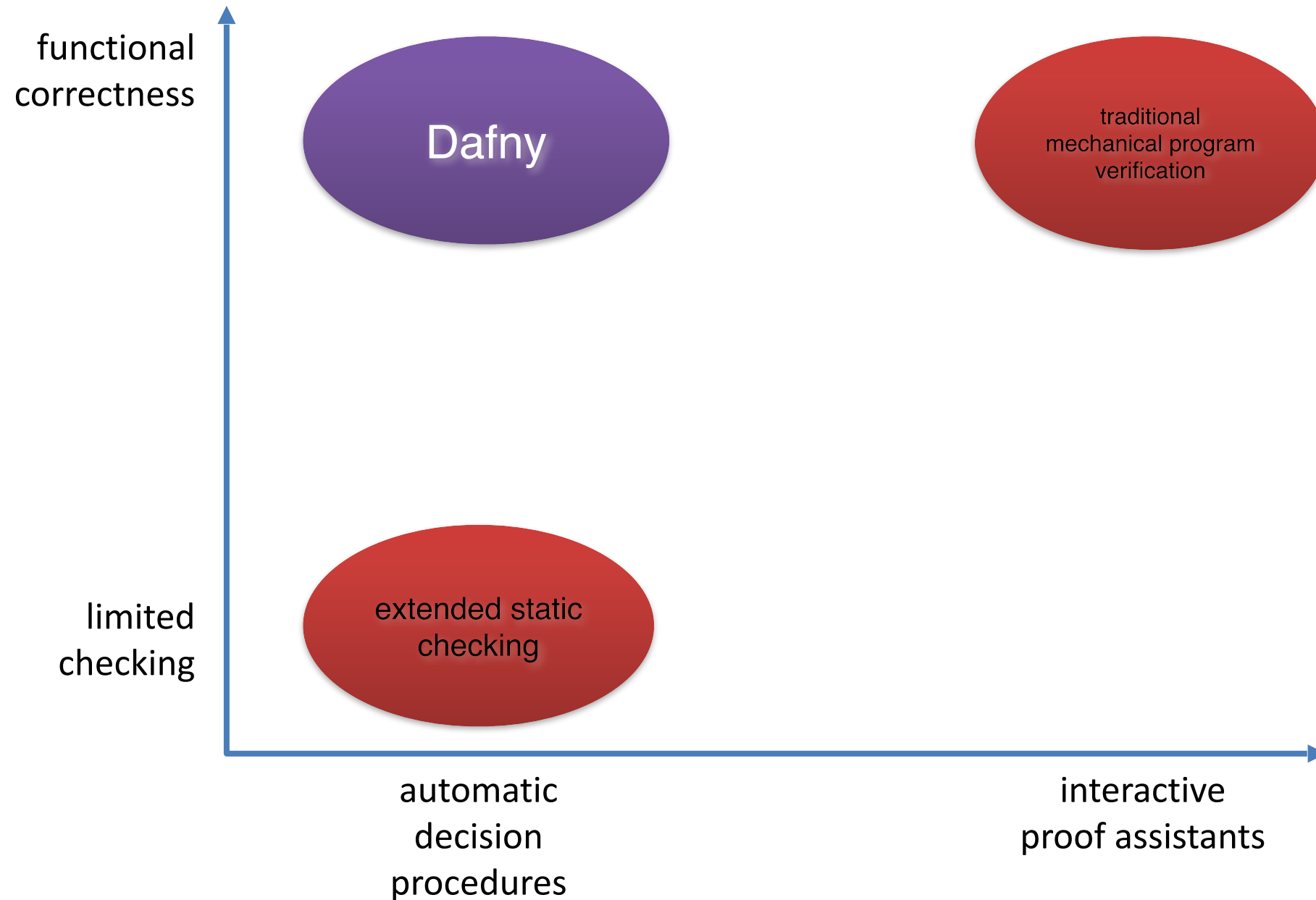
- Applies Hoare reasoning to programs
- User provides specifications in the form of pre- and postconditions, along with other assertions
- Dafny verifies that the program meets the specification
 - ▶ When successful, Dafny guarantees (total) functional correctness of the program

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, ...)

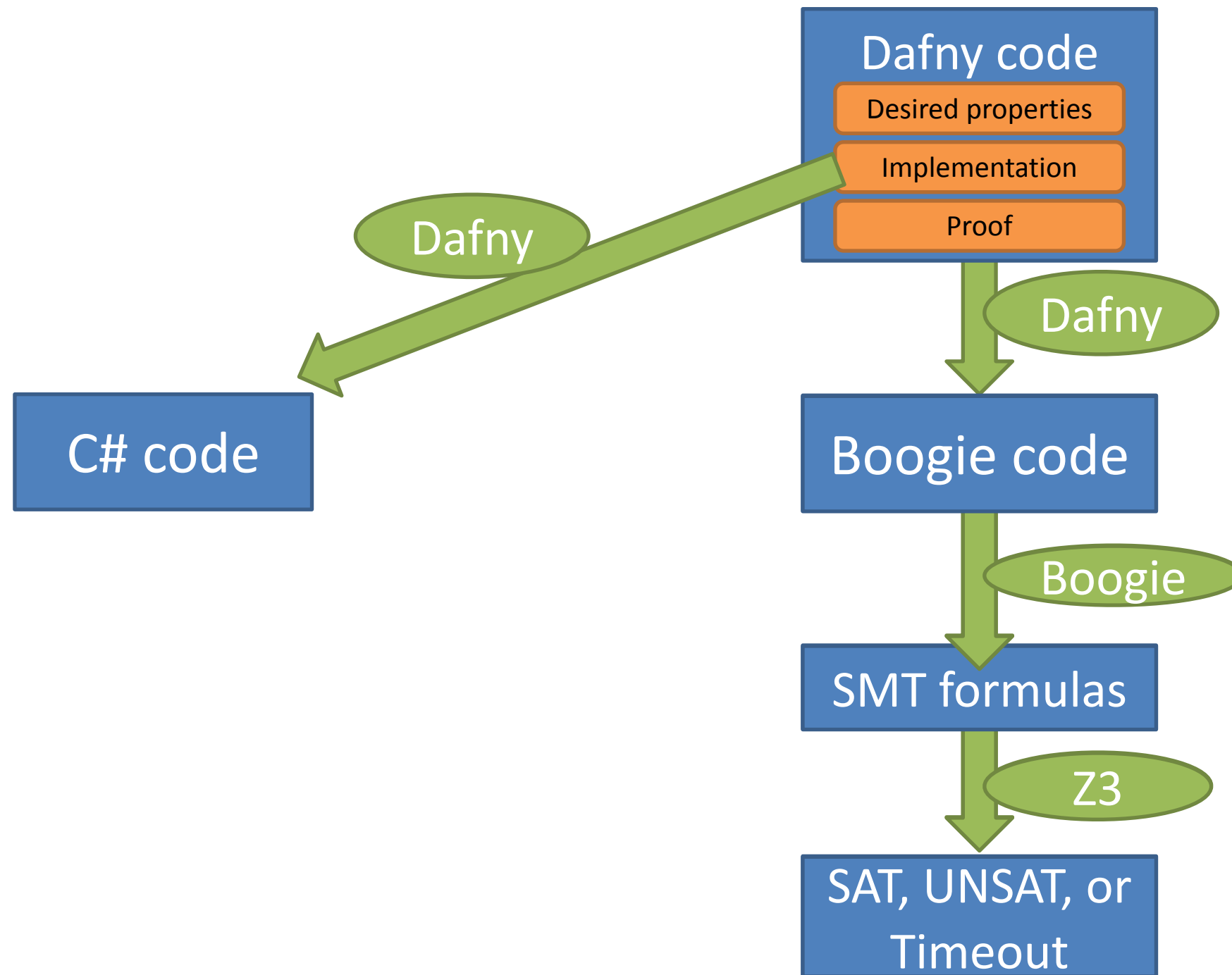
Types of Program Verification

15



Architecture

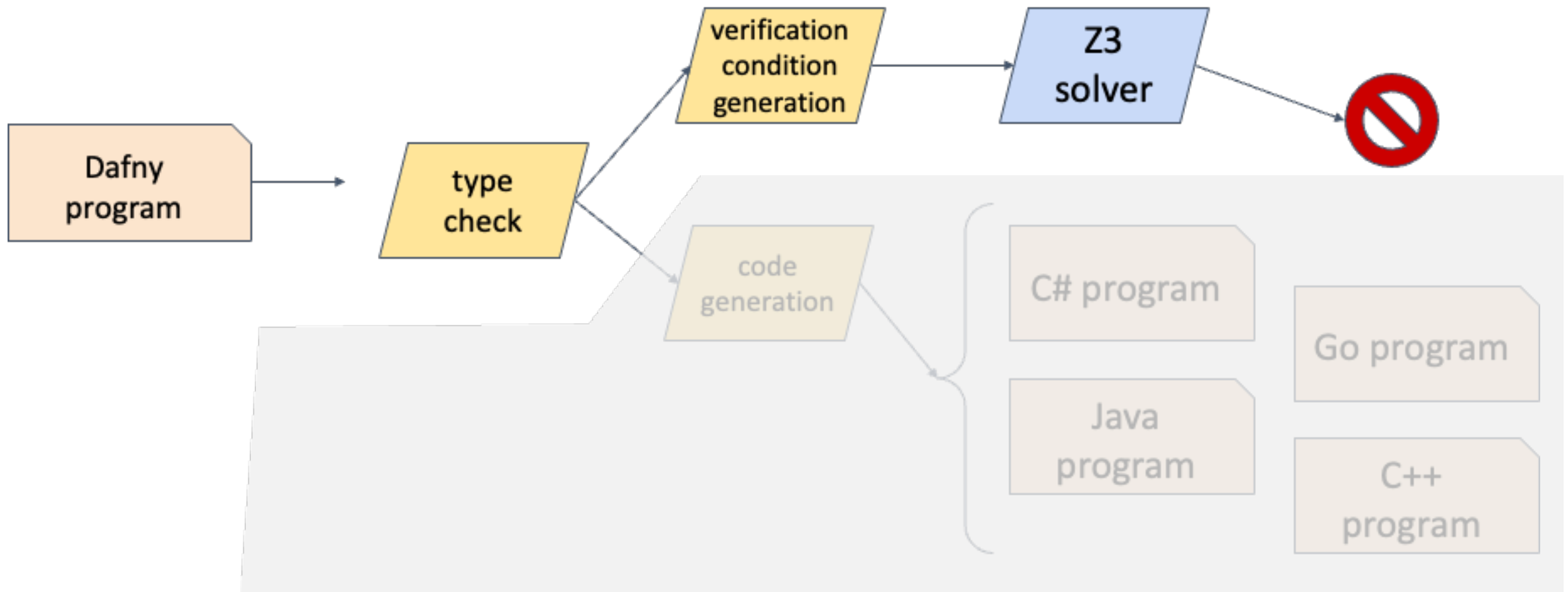
16



Dafny's architecture

Pipeline

17



Specifications

18

- 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

19

```
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

20

```
// 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

21

```
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

22

- 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

23

```
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

24

- 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

25

```
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

26

- 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 compiled 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

27

- `assert E` is a no-op if E holds, otherwise program faults.
- To show postcondition Q holds, i.e.,
 $WP(\text{assert } E, Q)$
 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

28

```
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

29

```
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

30

```
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; }
  }
```