The use and anatomy of an auto-active verifier

K. Rustan M. Leino

Harmonic

30 June – 3 July 2026
OPLSS 2026
Eugene, OR, USA

Auto-matic inter-active

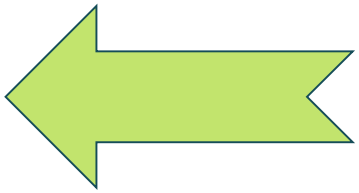
Program

```
method M(x: int)
  requires 0 ≤ x
  ensures Q(x)
{
  var n := 0;
  while n < x
    invariant Q(n)
  {
    …
  }
}
```

generates

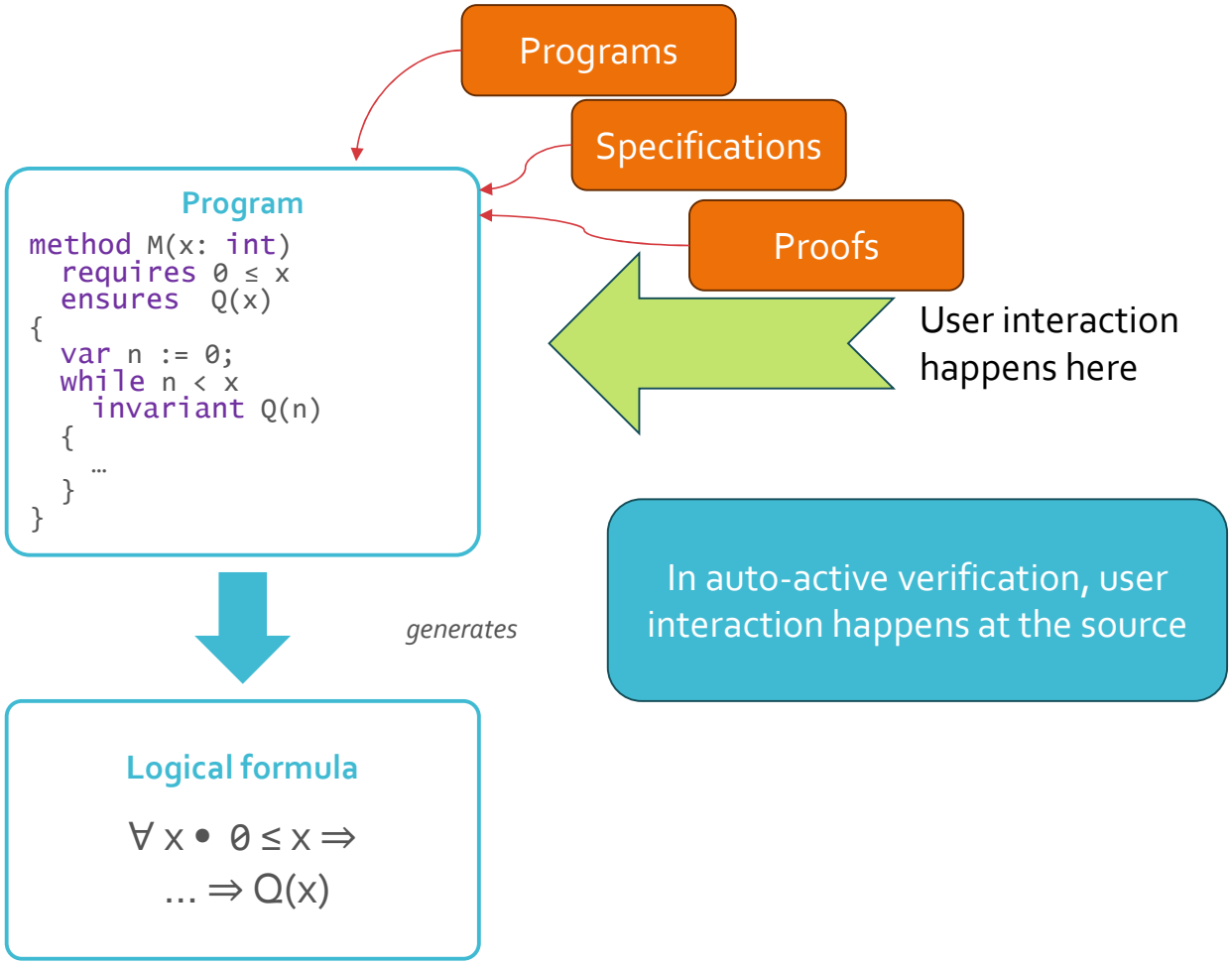
Logical formula

$\forall x \bullet 0 \le x \Rightarrow$
$\ldots \Rightarrow Q(x)$



User interaction happens here

Auto-matic inter-active



Outline

- Lecture 0: Small programs and proofs
- Lecture 1: Slighly larger programs and proofs
- Lecture 2: Implementation of a verifier
- Lecture 3: Reflections on language design

Goals of lectures 0 and 1

Teach concepts of program verification

Give you confidence that you can verify programs and write mechanically checked proofs

Show the auto-active interaction style

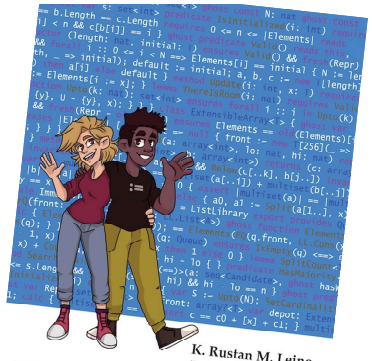
Dafny: language

- Dafny (`dafny.org`)
- Programming language
  - General purpose
  - Designed to support verification (proof-oriented)
  - Designed to be familiar to mainstream developers
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs

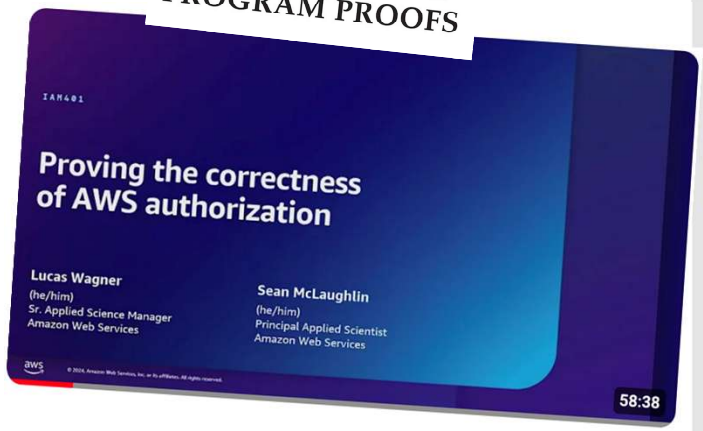
LLMs do well with Dafny

Some uses of Dafny

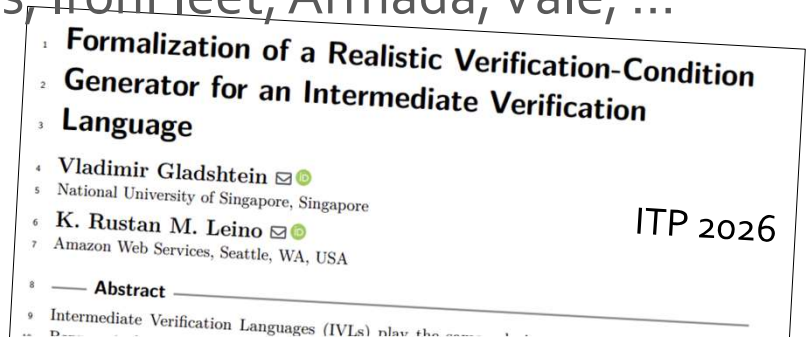
- Teaching
  - Over a decade, many universities



- Industrial use
  - AWS's authorization engine
    Paper at ICSE 2025.
    Talk at AWS re:Inforce 2024:
    https://youtu.be/oshxAJGrwMU



- Research projects
  - Ironclad Apps, IronFleet, Armada, Vale, …
  - Meta theory


Formalization of a Realistic Verification-Condition Generator for an Intermediate Verification Language

Vladimir Gladshtein
National University of Singapore, Singapore
K. Rustan M. Leino
Amazon Web Services, Seattle, WA, USA

Abstract

Intermediate Verification Languages (IVLs) play the

ITP 2026

Programs

- Double
- Min
- Sums
- BoolSort

"What remains to be done" invariants

```
function Sum(s: seq<int>): int {
  if |s| == 0 then 0 else s[0] + Sum(s[1..])
}


method ComputeSum(s: seq<int>) returns (r: int)
  ensures r == Sum(s)
{
  r := 0;
  var i := 0;
  while i < |s|
    invariant 0 ≤ i ≤ |s|
    invariant r == Sum(s[..i])
    invariant Sum(s) == r + Sum(s[i..])
  {
    r, i := r + s[i], i + 1;
  }
}
```

Sort with boolean keys: solution (variation 0)

```
method BoolSort(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;
    }
  }
}
```

Sort with boolean keys: solution (variation 1)

```
method BoolSort(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];
    }
  }
}
```

Sort with boolean keys: solution (variation 2)

```
method BoolSort(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];
    }
  }
}
```

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>`
`}`
(`else {` … `}` 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>`

Programs

- BruteSort
- Gauss
- Lists

BruteSort
(0/2)

```
method BruteSort(a: array<bool>)
  modifies a
  ensures forall i, j :: 0 ≤ i < j < a.Length ⟹ a[i] ⟹ a[j]
  ensures multiset(a[..]) == multiset(old(a[..]))
{
  var c0 := 0;
  ghost var c1 := 0;
  ghost var m := multiset{};
  var n := 0;
  while n < a.Length
    invariant 0 ≤ n ≤ a.Length
    invariant m == multiset(a[..n])
    invariant c0 == m[false] && c1 == m[true]
    invariant c0 + c1 == n
  {
    var x := a[n];
    if x {
      c1 := c1 + 1;
    } else {
      c0 := c0 + 1;
    }
    m := m + multiset{x};
    n := n + 1;
  }
  assert a[..] == a[..n];
  ⋮
```

BruteSort
(1/2)

```
  ⋮
  n := 0;
  while n < a.Length
    invariant 0 ≤ n ≤ a.Length
    invariant forall i, j :: 0 ≤ i < j < n ⟹ a[i] ⟹ a[j]
    invariant c0 ≠ 0 ⟹ forall i :: 0 ≤ i < n ⟹ !a[i]
    invariant multiset(old(a[..])) == m + multiset(a[..n])
    invariant c0 == m[false] && c1 == m[true]
    invariant c0 + c1 + n == a.Length
  {
    var x := c0 == 0;
    if x {
      c1 := c1 - 1;
    } else {
      c0 := c0 - 1;
    }
    m := m - multiset{x};
    a[n] := x;
    n := n + 1;
  }
  assert a[..] == a[..n];
}
```

Methods and lemmas

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

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

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

Methods and lemmas

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

Methods and lemmas

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

Methods and lemmas

```
lemma Gauss(n: nat)

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

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

    invariant TriangleSum(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!

Triangle numbers

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

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

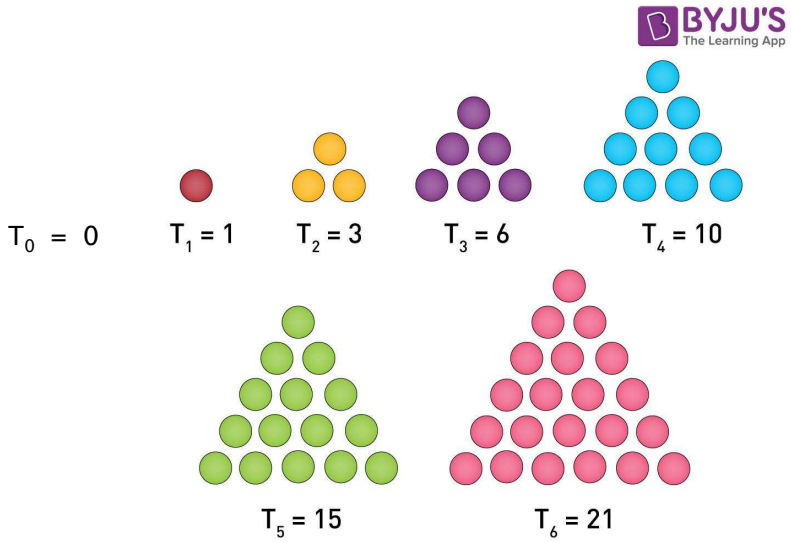
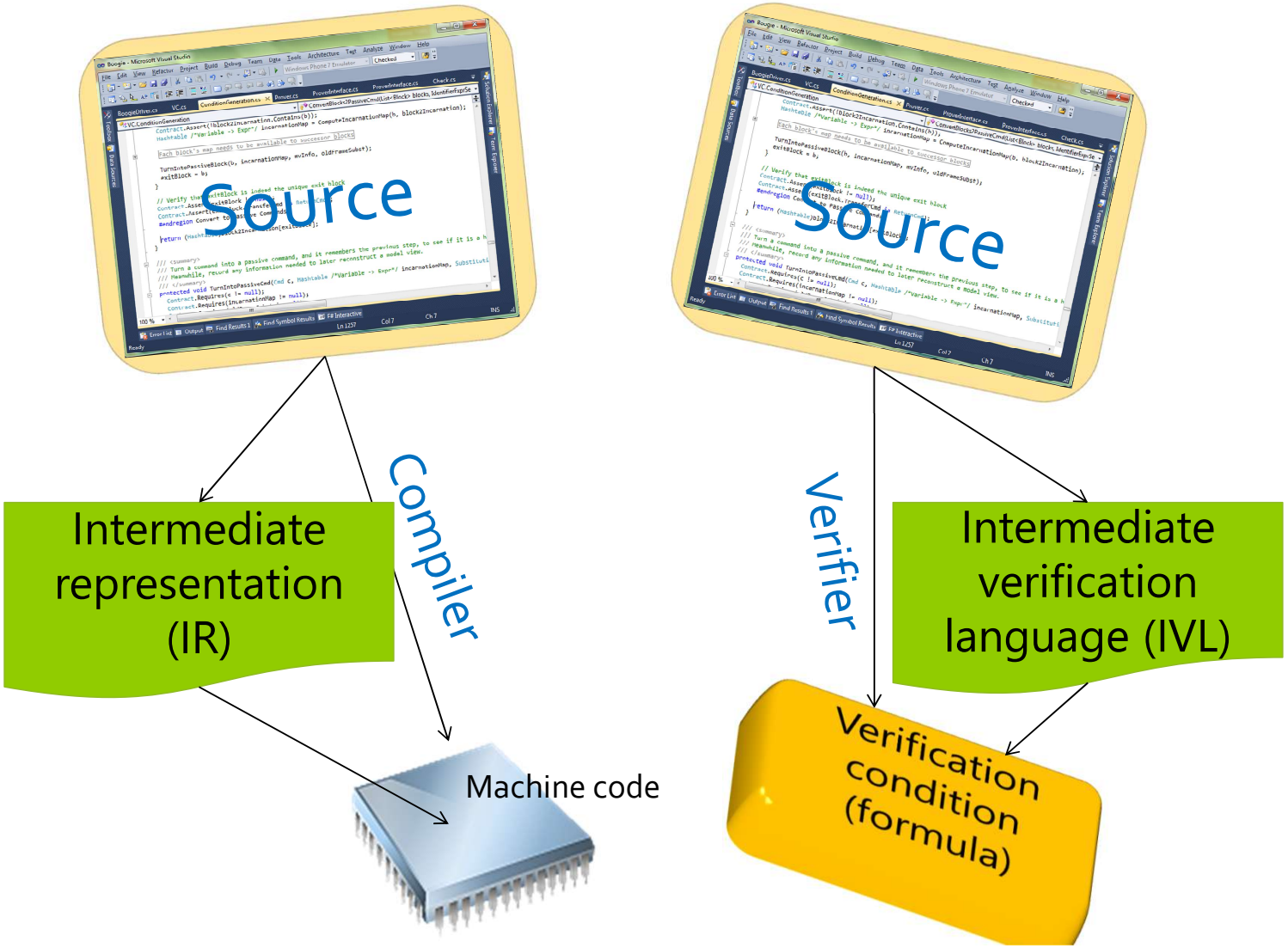


Image credit: byjus.com

Homework

- Change the second loop in `BruteSort` to go from `a.Length` down to `0`

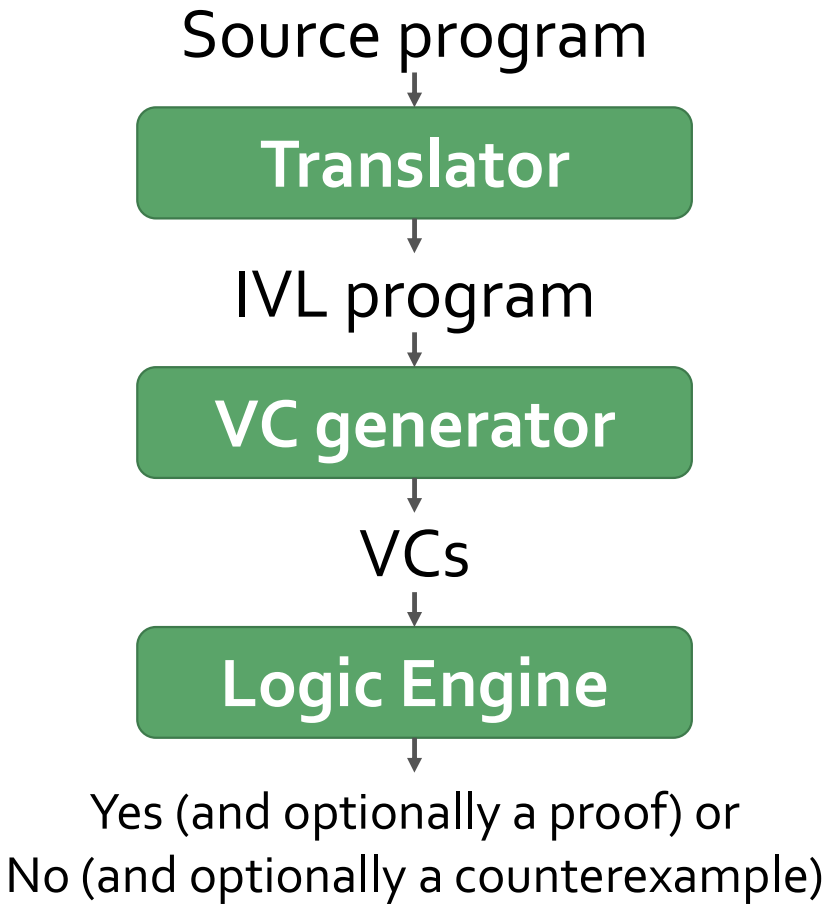
- Define a datatype `Tree` and functions `Depth` and `Mirror`. Then, prove
  - `Depth(Mirror(t)) == Depth(t)`
  - `Mirror(Mirror(t)) == t`


Source
Source
Compiler
Verifier
Intermediate representation (IR)
Intermediate verification language (IVL)
Machine code
Verification condition (formula)

Intermediate verification languages (IVLs)

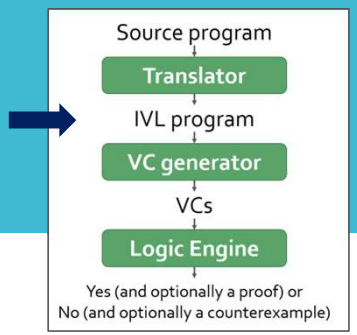
- Why3
- Boogie
- Viper
- Coma
- B3
- …

Verifier architecture



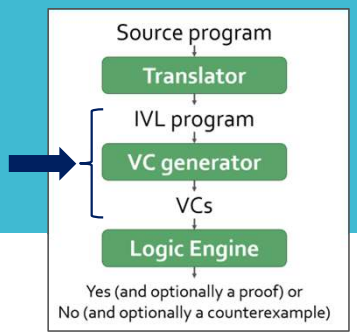
IVL = Intermediate Verification Language
VC = Verification Condition

IVL in a nutshell




```
IVL ::= Axiom* Proc*

Axiom ::= axiom Expr

Proc ::= procedure Id
           requires Expr
           ensures Expr
         { Stmt }
```

VC generation in a nutshell



Given

IVL

```
axiom A0
axiom A1
 ⋮
procedure M0 requires P0 ensures Q0 { S0 }
procedure M1 requires P1 ensures Q1 { S1 }
 ⋮
```

generate

VC

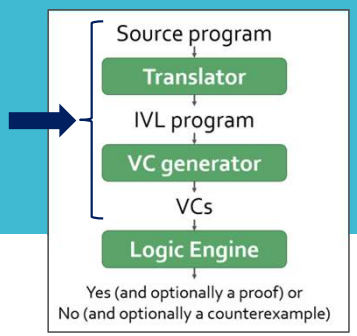
$A_0 \wedge A_1 \wedge \ldots \wedge P_0 \implies wp[\![S_0, Q_0]\!]$

$A_0 \wedge A_1 \wedge \ldots \wedge P_1 \implies wp[\![S_1, Q_1]\!]$

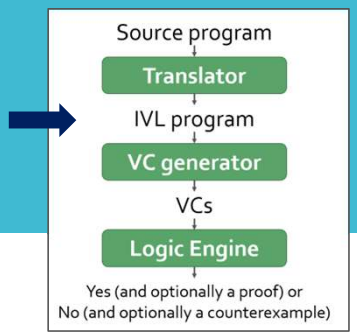
⋮

Demo

Source → IVL → VCs

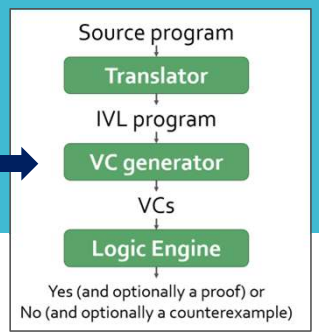


IVL in more detail




```
Stmt ::=
    |   var x := Rhs
    |   x := Rhs
    |   assert Expr
    |   Stmt ; Stmt
    |   Stmt [] Stmt
    |   loop
    |   procedure call
    |   …

Rhs ::=  Expr | *
```

Weakest preconditions



$$wp[\![\ x := e,\ Q\ ]\!] = Q[x := e]$$

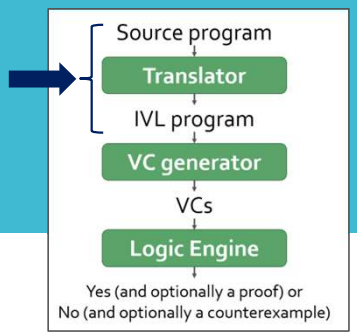
$$wp[\![\ x := *,\ Q\ ]\!] = \forall x.\ Q$$

$$wp[\![\ \texttt{assert}\ e,\ Q\ ]\!] = e \wedge Q$$

$$wp[\![\ S;T,\ Q\ ]\!] = wp[\![\ S,\ wp[\![\ T,\ Q\ ]\!]\ ]\!]$$

$$wp[\![\ S \,[]\, T,\ Q\ ]\!] = wp[\![\ S,\ Q\ ]\!] \wedge wp[\![\ T,\ Q\ ]\!]$$

Well-formedness (simple)



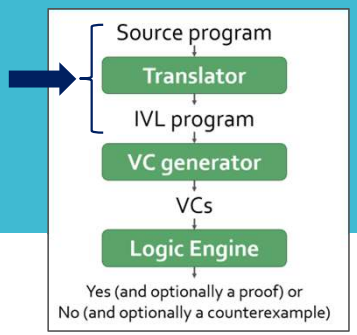
Source

```
x := a / b
```

IVL

```
assert b ≠ 0
x := a / b
```

Choice of integer treatments



Source

```
var a: u32, b: u32, x: u32
…
x := a + b
```

IVL

```
x := a + b
```

IVL

```
x := (a + b) % 0x1_0000_0000
```

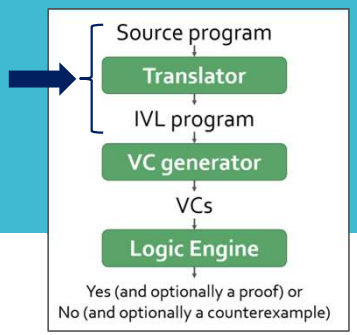
IVL

```
assert a + b < 0x1_0000_0000
x := a + b
```

IVL

```
if a + b < 0x1_0000_0000 {
  x := a + b
} else {
  model the throwing of an exception
}
```

Definite assignment



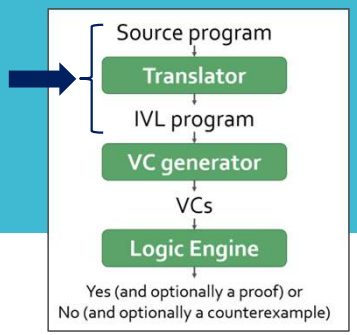
Source

```
var x
if B {
  x := 100
}
…
if C {
  y := x
}
```

IVL

```
var x := *;  var _defined_x := false
if B {
  x := 100;  _defined_x := true
}
…
if C {
  assert _defined_x;  y := x
}
```

Translating a type into a larger type



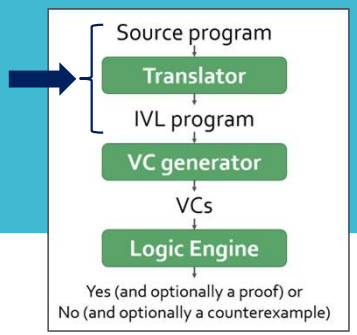
Source

```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
if c == Red {
  S0
} else if !(c == Red) && c == Blue {
  S1
} else if !(c == Red) && !(c == Blue) &&
          c == Green {
  S2
} else {
  assert false;
}
```

Translating a type into a larger type



Source

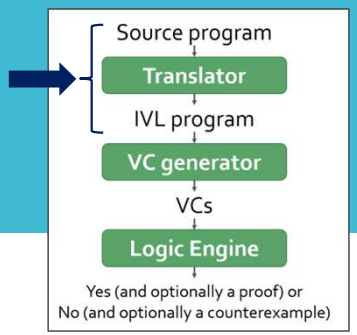
```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
if c == Red {
  S0
} else if c == Blue {
  S1
} else if c == Green {
  S2
} else {
  assert false;
}
```

But the source-language semantics guarantees that control never reaches this branch

Translating a type into a larger type



Source

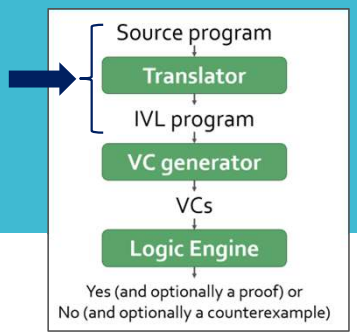
```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
if c == Red {
  S0
} else if c == Blue {
  S1
} else if c == Green {
  S2
} else {
  assume we never get here
}
```

We would rather do this

Translating a type into a larger type



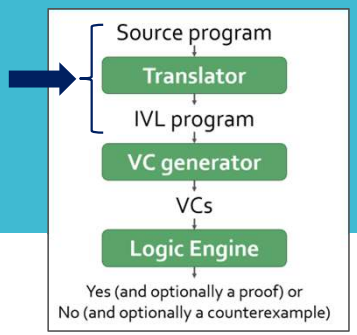
Source

```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
if c == Red {
  S0
} else if c == Blue {
  S1
} else if c == Green {
  S2
} else {
  assume false;
}
```

Translating a type into a larger type



Source

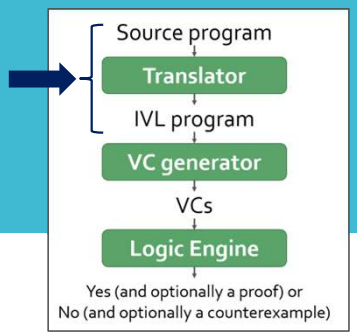
```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
  assume c == Red;   S0
▯
  assume c == Blue;   S1
▯
  assume c == Green;   S2
▯
  assume false
```

It turns out that `assume false` is the unit element of ▯

Translating a type into a larger type



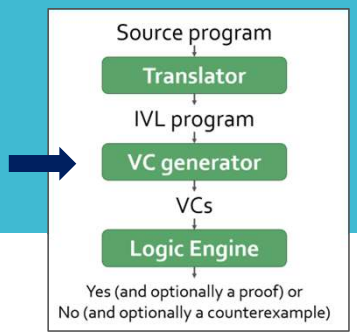
Source

```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

IVL

```
  assume c == Red;  S0
[]
  assume c == Blue;  S1
[]
  assume c == Green;  S2
```

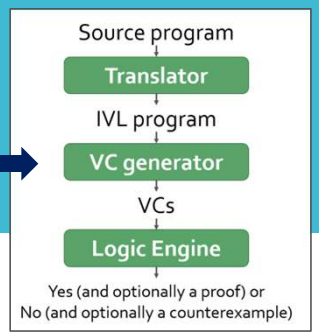
Weakest preconditions: assume



$$wp[\![\ \texttt{assert}\ e,\ Q\ ]\!] = e \wedge Q$$

$$wp[\![\ \texttt{assume}\ e,\ Q\ ]\!] = e \Longrightarrow Q$$

Desugar `call`



IVL

```
procedure M(x) returns (y)
  requires P
  ensures Q


call a := M(b)
```

Desugared IVL

```
{
  var x := b
  assert P
  var y := *
  assume Q
  a := y
}
```

Bind formal in-parameter

Check precondition

Effect the procedure's effect

Assign actual out-parameter

Desugar loop

Note that this translation also allows break statements in S

IVL

```
while e
  invariant J
{
  S
}
```

let x denote the variables assigned to by S

Desugared IVL

```
assert J
```

Check the invariant on entry

```
x := *
assume J
```

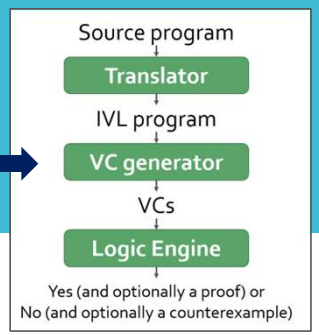
Fast forward to an arbitrary loop iteration

```
if e {
  S
  assert J
  assume false
}
```

Start in on the next iteration

Check invariant is maintained

Stop checking



wp of loop desugaring

wp⟦ `assert J; x := *; assume J;`
`if e { S; assert J; assume false; }`, Q ⟧
=
wp⟦ `assert J; x := *; assume J`,
wp⟦ `if e { S; assert J; assume false; }`, Q ⟧⟧
=
wp⟦ `assert J; x := *; assume J`,
(e ⟹ wp⟦ `S; assert J; assume false`, Q ⟧) ∧
(¬e ⟹ Q) ⟧
=
wp⟦ `assert J; x := *; assume J`,
(e ⟹ wp⟦ `S; assert J`, false ⟹ Q ⟧) ∧
(¬e ⟹ Q) ⟧
=
wp⟦ `assert J; x := *; assume J`,
(e ⟹ wp⟦ `S; assert J`, true ⟧) ∧
(¬e ⟹ Q) ⟧
=
wp⟦ `assert J; x := *; assume J`,
(e ⟹ wp⟦ `S`, J ∧ true ⟧) ∧
(¬e ⟹ Q) ⟧
=

wp⟦ `assert J; x := *; assume J`,
(e ⟹ wp⟦ `S`, J ⟧) ∧ (¬e ⟹ Q) ⟧
=
wp⟦ `assert J; x := *`,
J ⟹ (e ⟹ wp⟦ `S`, J ⟧) ∧ (¬e ⟹ Q) ⟧
=
wp⟦ `assert J`,
∀ x. J ⟹ (e ⟹ wp⟦ `S`, J ⟧) ∧ (¬e ⟹ Q) ⟧
=
J ∧ ∀ x. J ⟹ (e ⟹ wp⟦ `S`, J ⟧) ∧ (¬e ⟹ Q)
=

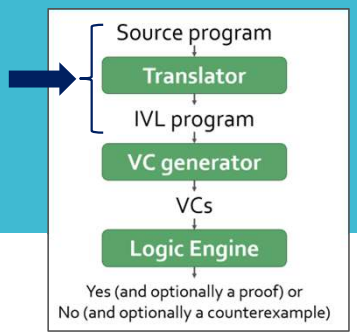
J ∧
(∀ x. J ∧ e ⟹ wp⟦ `S`, J ⟧) ∧
(∀ x. J ∧ ¬e ⟹ Q)

Hoare-logic rule

$$\frac{\{J \wedge e\}\ S\ \{J\}}{\{J\}\ \text{while } e \text{ do } S\ \{J \wedge \neg e\}}$$

$J \wedge$
$(\forall x.\ J \wedge e \Longrightarrow wp[\![S, J]\!]) \wedge$
$(\forall x.\ J \wedge \neg e \Longrightarrow Q)$

Translating `calc`



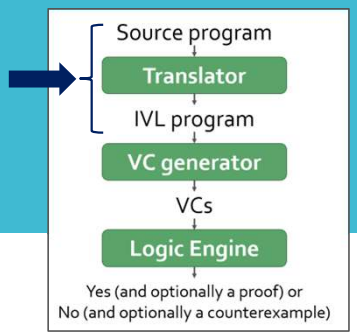
Source

```
calc {
  E0;
==  { S0 }
  E1;
<   { S1 }
  E2;
}
```

IVL

```
S0
assert E0 == E1

S1
assert E1 < E2

assume E0 < E2
```

Translating `calc`



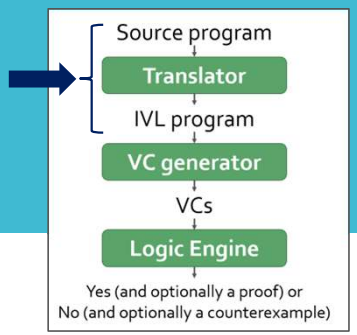
Source

```
calc {
  E0;
==  { S0 }
  E1;
<   { S1 }
  E2;
}
```

IVL

```
  S0
  assert E0 == E1
  assume false
□
  S1
  assert E1 < E2
  assume false
□
  assume E0 < E2
```

Translating `calc`



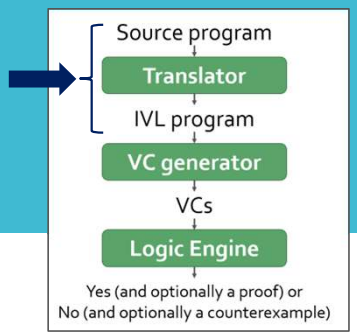
Source

```
calc {
  E0;
==  { S0 }
  E1;
<   { S1 }
  E2;
}
```

IVL

```
  S0; assert wf⟦ E0 ⟧; assert wf⟦ E1 ⟧
  assert E0 == E1
  assume false
[]
  S1; assert wf⟦ E2 ⟧
  assert E1 < E2
  assume false
[]
  assume E0 < E2
```

Translating `calc`



Source

```
calc {
  E0;
==  { S0 }
  E1;
<   { S1 }
  E2;
}
```

IVL

```
  S0; assert wf⟦ E0 ⟧; assert wf⟦ E1 ⟧
  assert E0 == E1
  assume false
□
  S1; assume wf⟦ E1 ⟧; assert wf⟦ E2 ⟧
  assert E1 < E2
  assume false
□
  assume wf⟦ E0 ⟧ ∧ wf⟦ E2 ⟧ ∧ E0 < E2
```

Some references

(the first 3 describe good ways to compute wp, the last shows the entire 2008 translation from Dafny to Boogie)

- **Avoiding exponential explosion: generating compact verification conditions**
  Cormac Flanagan and James B. Saxe.
  POPL 2001: 193-205.
- **Efficient weakest preconditions**
  K. Rustan M. Leino
  Information Processing Letters, 93(6), 2005.
  https://leino.science/papers/krml114.pdf
- **Weakest-Precondition of Unstructured Programs**
  Mike Barnett and K. Rustan M. Leino.
  PASTE 2005.
  https://leino.science/papers/krml157.pdf
- **Specification and verification of object-oriented software**
  K. Rustan M. Leino.
  Marktoberdorf International Summer School 2008, lecture notes.
  https://leino.science/papers/krml190.pdf

Logic engine

- Satisfiability modulo theories (SMT) solver
- Saturation-based first-order prover
- AI

Reflections on language design

Lecture 3

Learning from trying to verify Java and C#

- Strings

String concatenation in Java

```
String concat(String s, String t)
{
    return s + t;
}
```

Pure methods

```
@Pure
String concat(String s, String t)
{
    return s + t;
}


assert concat(a, b) == concat(a, b);
```

Learning from trying to verify Java and C#

- Strings and other immutable types should not be references
  - Include non-reference types in the language
- "Pure" methods are hard to make pure enough (that is, too few methods are actually pure)
  - Provide functions
- `Equals` methods are the result of a too-OO mindset
  - Use references if you need mutation or identity
  - Otherwise, use datatypes
- Don't give access to `this` until it's really constructed
  - Use 2-phase constructors
- It takes effort to prove the absence of arithmetic overflows
  - Provide easy access to unbounded integers (even the default?)

Syntax

- `function`
- `ghost function`

Specifications

- Sometimes, loops are overly specific

Initialize a matrix

```
method Init<X>(arr: array2<X>, x: X)
  modifies arr
  ensures forall i, j :: 0 ≤ i < arr.Length0 &&
                         0 ≤ j < arr.Length1 ⟹
                         arr[i, j] == x
{
 var m := 0;
  while m < arr.Length0
    invariant 0 ≤ m ≤ arr.Length0
    invariant forall a, b ::
      0 ≤ a < m && 0 ≤ b < arr.Length1 ⟹ arr[a, b] == x
  {
    var n := 0;
    while n < arr.Length1
      invariant 0 ≤ n ≤ arr.Length1
      invariant forall a, b ::
        0 ≤ a < m && 0 ≤ b < arr.Length1 ⟹ arr[a, b] == x
      invariant forall b :: 0 ≤ b < n ⟹ arr[m, b] == x
    {
      arr[m, n] := x;
      n := n + 1;
    }
    m := m + 1;
  }
}
```

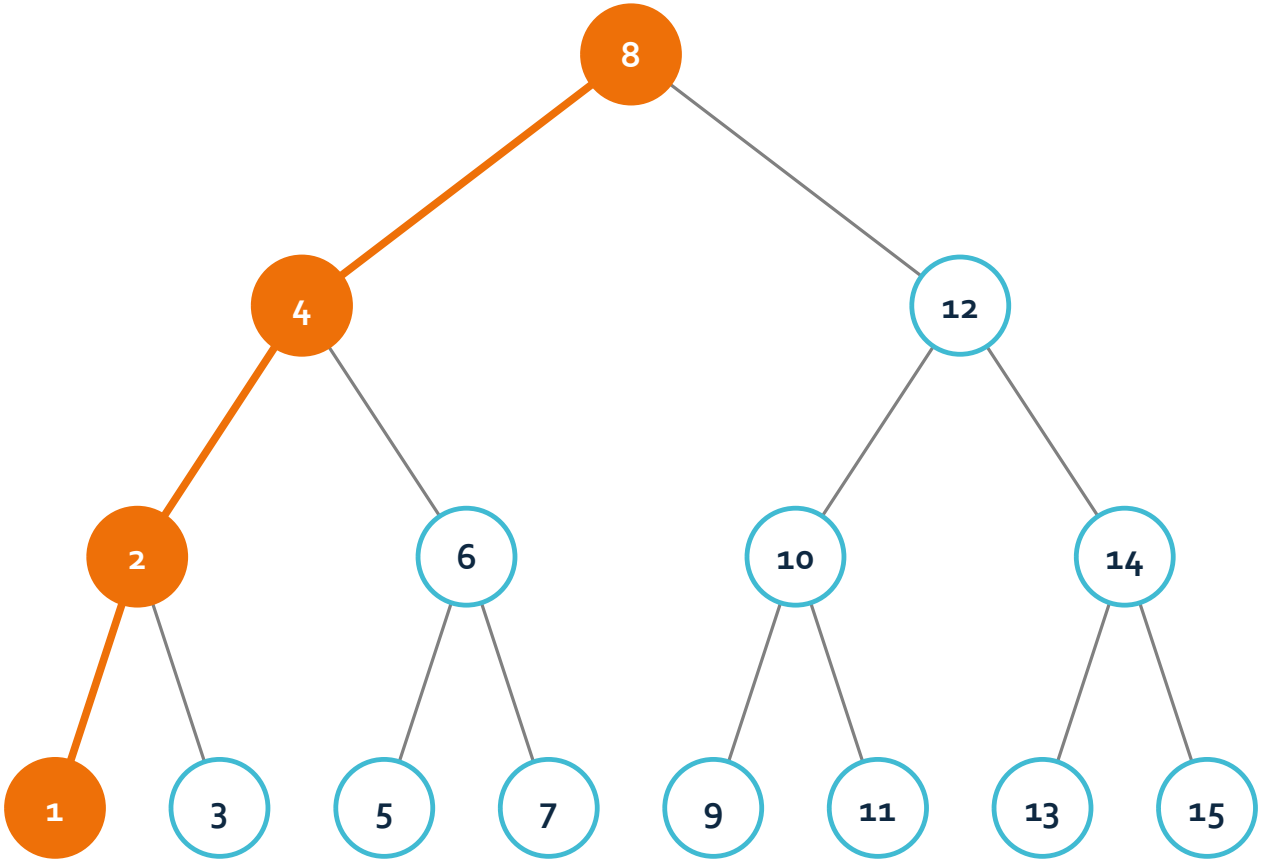
Aggregate statement (forall statement)

```
method Init<X>(arr: array2<X>, x: X)
  modifies arr
  ensures forall i, j :: 0 ≤ i < arr.Length0 &&
                         0 ≤ j < arr.Length1 ⟹
                         arr[i, j] == x
{
  forall i, j | 0 ≤ i < arr.Length0 && 0 ≤ j < arr.Length1
  {
    arr[i, j] := x;
  }
}
```

Specifications

- Sometimes, loops are overly specific
  - Add aggregate statements
- Some loops are difficult to specify

Remove the minimum of binary search tree



The minimum is the leftmost node: follow left children from the root, 8 → 4 → 2 → 1.

Code for removing minimum

```
class Node {
  var value: int
  var parent: Node
  var left: Node?
  var right: Node?
  ghost var Descendants: set<Node>
}
```

```
Remove(root: Node) {
  var p := root
  while p.left != null {
    p := p.left
  }
  p.parent.left := null
  // update all the Descendants
  return p
}
```

```
RecursiveRemove(root: Node) {
  if p.left == null {
    p.parent.left := null
    return p
  }
  var r := RecursiveRemove(p.left)
  p.Descendants := p.Descendants - {r}
  return r
}
```

*) specifications and various edge cases omitted

Specifications

- Sometimes, loops are overly specific
  - Add aggregate statements
- Some loops are difficult to specify
  - Offer tail-recursive compilation, *modulo ghosts*
- Having to rewrite functions with accumulators to make them tail recursive is cumbersome
  - Offer auto-accumulator tail recursion
- User-defined well-founded orders are difficult to verify correct
  - Provide a fixed well-founded order for all types

Framing

- Dafny supports specifying and verifying mutable data structures in the heap by
  - Including `modifies` and `reads` clauses
  - Including sets
  - Including ghosts

Conclusion

- Program proofs are becoming more accessible
- Become familiar with proving programs!
- Teach!

Program safely!