EECS498-003
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

Statically checking for correctness

What we want is a “static correctness check”, akin to a static type check

You write your code normally, but if you introduce bugs the checker will tell you

When the checker complains, you have to spend some time to convince it that your code is right---if indeed it is

Using a Theorem Prover

Express the execution of the system and its correctness as a mathematical formula (done automatically by the language)

Give the formula to a theorem prover, effectively asking:
”If the system behaves this way, is it possible for its correctness to be violated?”

A negative answer means the system is proven to be correct
A positive answer means there is still work to do, either:

- the system is indeed incorrect
- the proof is incomplete

Using Dafny

- We will be using Dafny as our verification language
- Dafny is an imperative language designed with formal verification in mind
  - …and plenty of functional language features
- Dafny uses an SMT solver (Z3) to automate verification to a large degree
  - …but it needs our help sometimes
- Most of the high-level skills are transferrable…
  - …but some are specific to Dafny and/or automation

Getting started with Dafny

- In the lab on Friday, Keshav will go over instructions for installing Dafny 4.4

- The simplest way to use Dafny is via the Visual Studio plugin
  - Gives you a nice interface

- You can also invoke Dafny on the command line:
  - dafny myFile.dfy

Dafny in Docker

- We provide you with a Docker container that has Dafny pre-installed
  - Makes it easy to get started
  - Ensures everyone is using the same Dafny version as the autograder
  - Not highly recommended for the bulk of your development
- Download and run it like this:
  - `docker pull ekaprits/eecs498-009:latest`
  - `docker container run --mount src=$PWD,target=/home/autograder/working_dir,type=bind,readonly -t -i ekaprits/eecs498-009:latest`
- CAEN machines have some partial support for Docker
  - If you don’t have access to a machine that can run Docker, contact me ASAP

Administrivia

- Please remember to upload your picture, if you haven’t
  - https://verification.eecs.umich.edu/self.php
- Lab is tomorrow, Friday 9:30-11:30 in GGBL 2147
- See Piazza post for a research opportunity on a project with Max New and Xinyu Wang

Learning Dafny

We will be using Dafny as our verification language

Dafny is a programming language built with verification in mind

- It supports both imperative and declarative programming styles

Imperative vs declarative

Imperative style

Here's what I want you to **do**

```
upper_bound = 0;
for item in list:
  if item > upper_bound:
    upper_bound = item;
return upper_bound
```

Python (imperative)

```
small_nums = []
for i in range(20):
  if i < 5:
    small_nums.append(i)
```

Declarative style

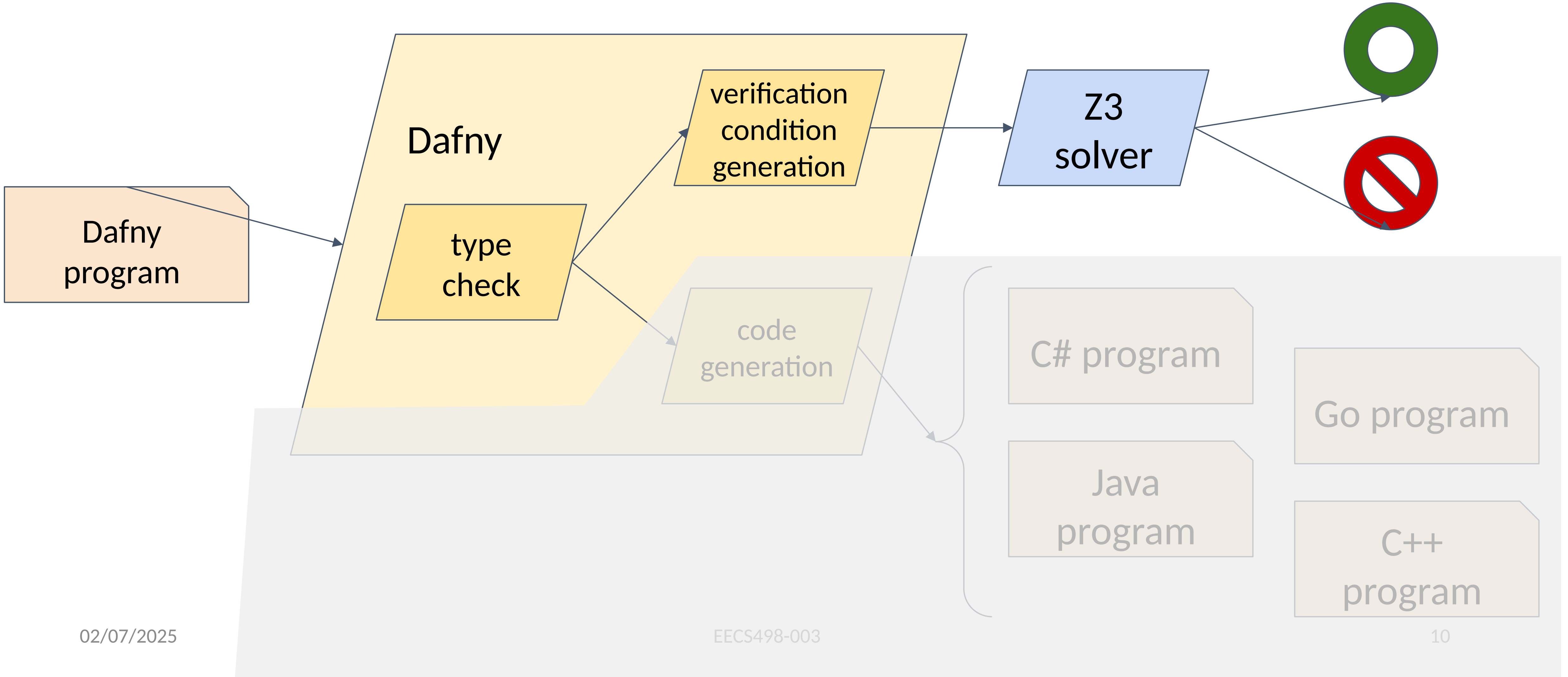
Here's what I want you to **return**

```
return upper_bound such that:
  forall item in list
    item <= upper_bound
```

Python (declarative)

```
small_nums = [x for x in range(20) if x < 5]
```

The Dafny pipeline



We will use the declarative parts of Dafny

Ignore the imperative parts (mostly)

- mutable objects
- heap “framing”: reads, modifies, fresh
- !new, ==

The declarative/mathematical/functional subset is most useful in writing high-level protocols and specifications

Running Dafny

- In Visual Studio: verification “onChange” or “onSave”

- On the command line:
  - `dafny /compile:0 /errorTrace:0 someDafnyFile.dfy`

Data constructs

This is a mathematical integer, not a machine integer

Basic primitives	int bool
Immutable compounds	set<T> seq<T> map<A, B> datatype
Mutable objects	class

Procedure-like constructs

As in math, not C:
- f(x, y) == f(x, y)
- definition substitution

A function returning bool

	expression context	*statement context*
ghost (not compiled)	ghost function ghost predicate	lemma
executable	function	method

Important difference: lemmas are opaque, while functions are not!

Function syntax

explicitly typed parameters

function result type

```
function eval_linear(m: int, b: int, x: int) : int
{
    m * x + b
}
```

definition body is an expression whose type matches result declaration

- `predicate` means “`function` returning `bool`”.

Lemma syntax

```
lemma MyFirstLemma(x: int)
{
  assert x >= 0;
  assert x >= -1;
}
```

definition body is an imperative-style statement context

assert() is a static check!

Dafny will attempt to prove the assertion. Regardless of the result, subsequent code will assume that x >= 0

Remember that lemmas are opaque!

Pre- and postconditions

```
lemma IntegerOrdering(a: int, b: int)
  requires b == a + 3
  ensures a < b
{
  assert a < b;
}
```

Precondition: statically checked anywhere this lemma is called

Postcondition: an exported assertion

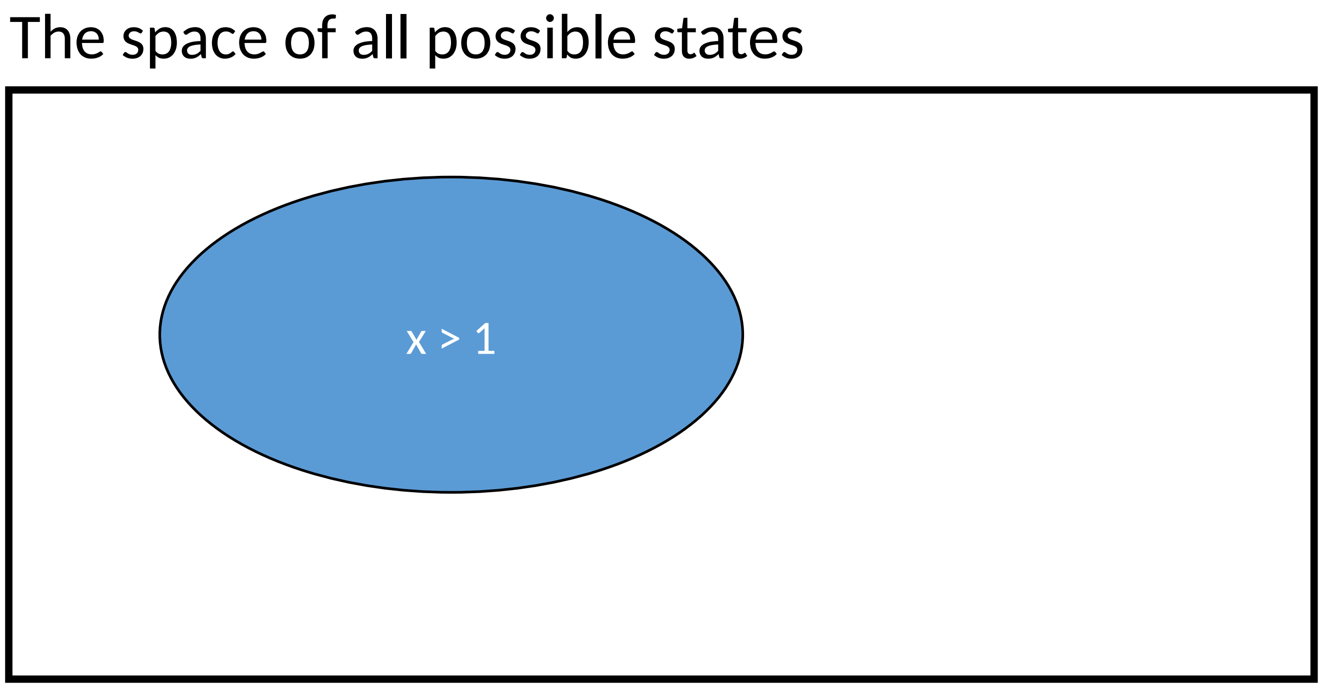
Pre- and postconditions

```
lemma IntegerOrdering(a: int, b: int)
           b == a + 3
      ==> a < b
{
  assert a < b;
}
```

Messing with preconditions

```
lemma IntegerOrdering(a: int, b: int)
  requires b == a + 3
  requires a > b + 1
  ensures a < b
{
  // proof goes here
}
```

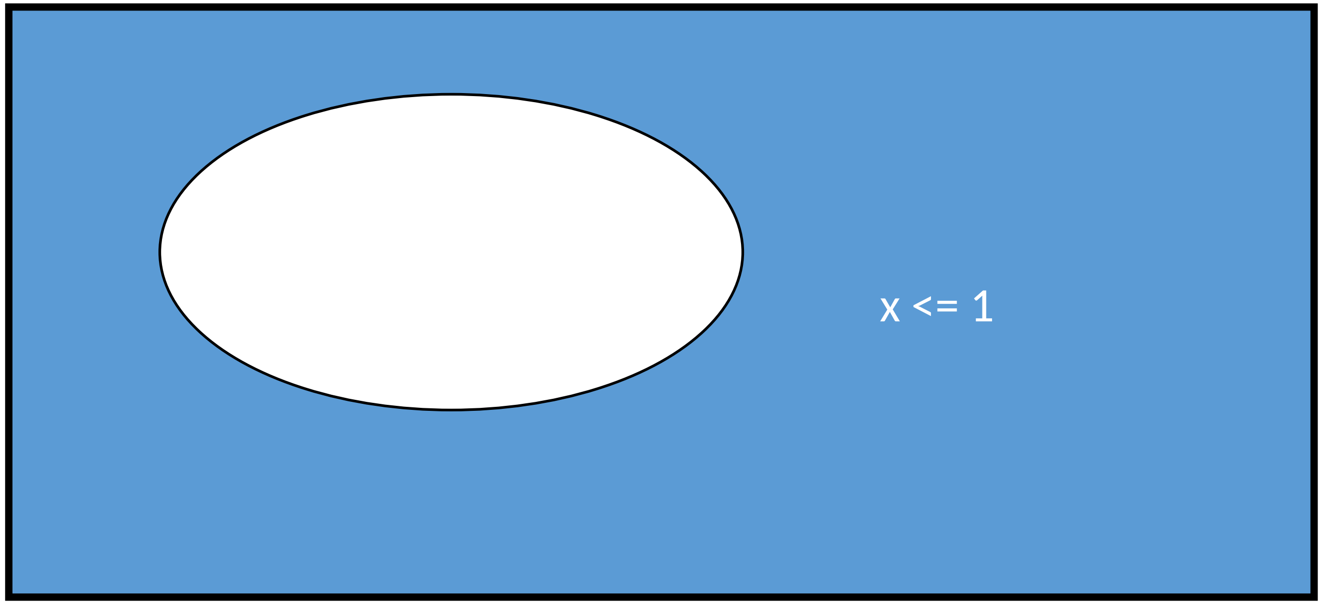
Predicates



A predicate (or any Boolean expression) is a *set of states*

Predicates

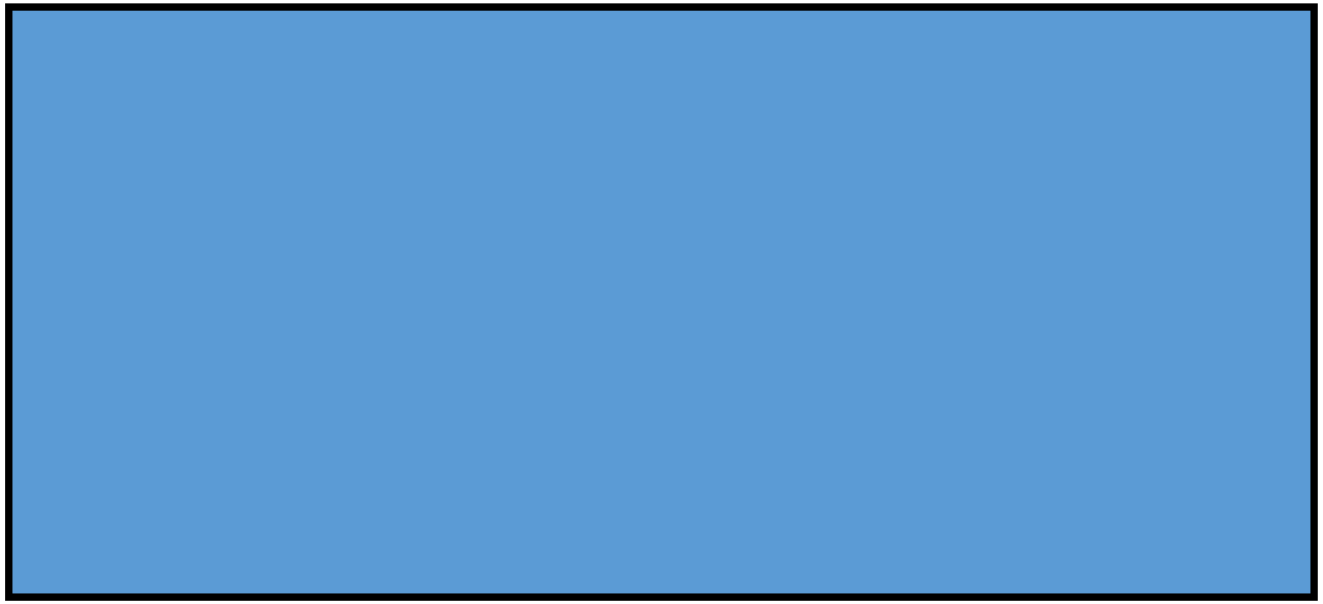
The space of all possible states



A predicate (or any Boolean expression) is a *set of states*

Predicates

The space of all possible states



What predicate (Boolean expression) is this?

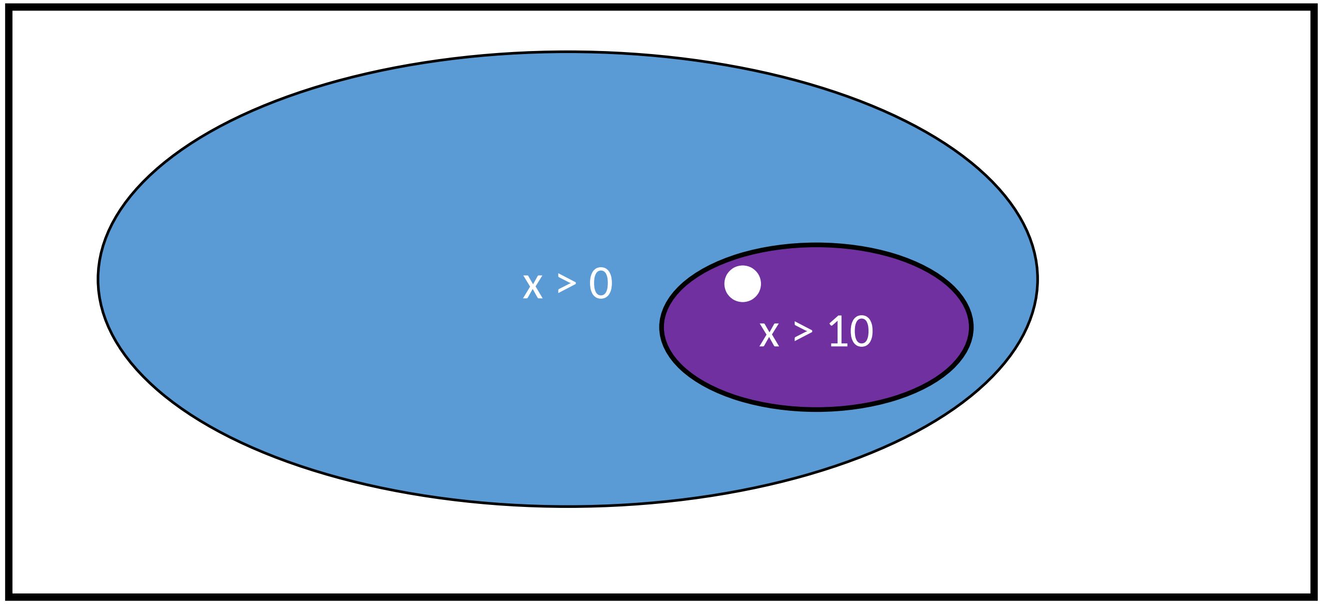
Predicates

The space of all possible states

What predicate (Boolean expression) is this?

Implications

The space of all possible states



What does an implication look like in this graph?

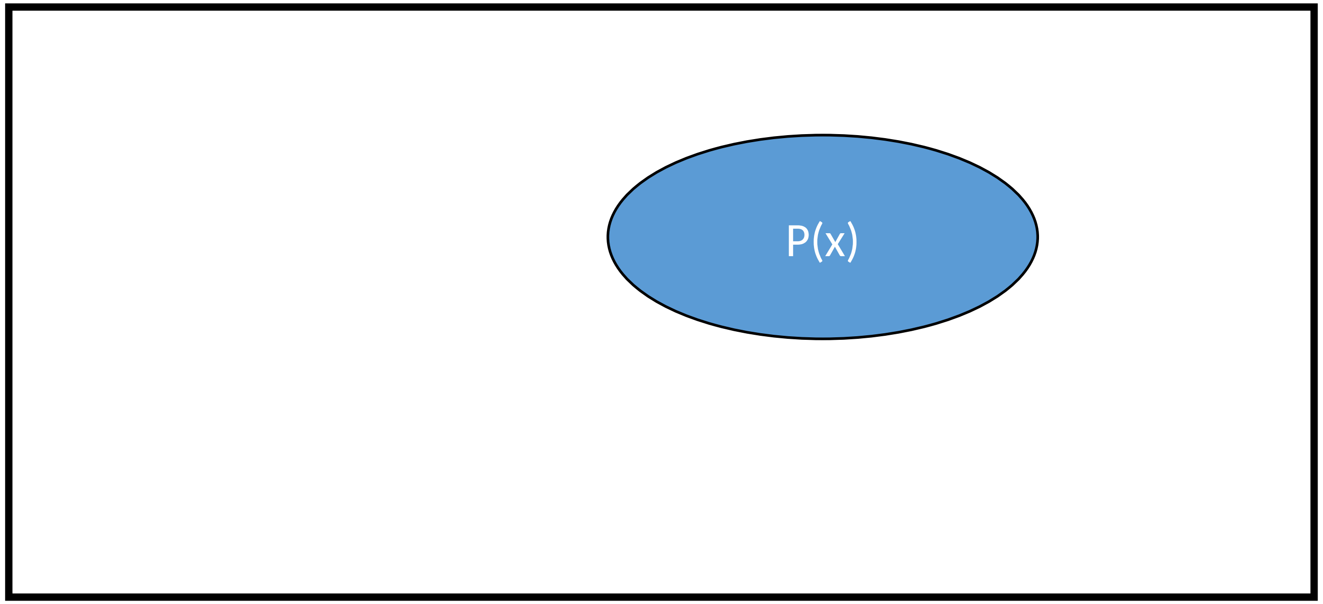
Logical statement:

```
x > 10 ==> x > 0
```

Visual equivalent:
If you belong in the “inner” predicate, you must also belong in the “outer” one

Reasoning about *false*

The space of all possible states



Does this hold?

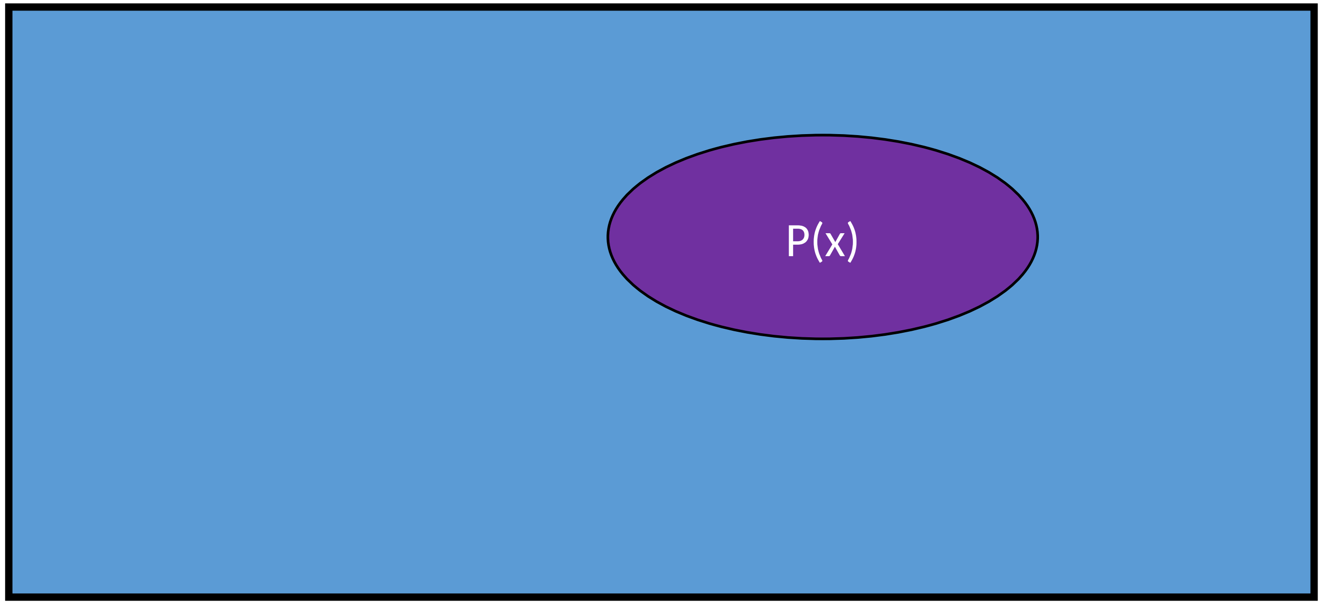
`false ==> P(x)`

Does this hold?

`P(x) ==> false`

Reasoning about *true*

The space of all possible states



Does this hold?

`true ==> P(x)`

Does this hold?

`P(x) ==> true`

Messing with preconditions

```
lemma IntegerOrdering(a: int, b: int)
  requires b == a + 3
  requires a > b + 1
  ensures a < b
{
  // proof goes here
}
```

Messing with postconditions

Is the following lemma ever useful?

```
lemma SomeLemma(x: int, y: int)
  requires P(x, y)
  ensures false
{
  // proof goes here
}
```

Opacity

```
ghost function eval_linear(m: int, b: int, x: int) :
int
{
    m * x + b
}
```

```
lemma zero_slope(m: int, b: int, x1: int, x2:int)
{
  if (m == 0) {
    assert eval_linear(m, b, x1) == eval_linear(m, b, x2);
  }
}
```

- This lemma verifies because it can see inside the definition of `function eval_linear()`
- …but lemma bodies are opaque! The result of this verification can't be used anywhere else.

Opacity

```
lemma zero_slope(m: int, b: int, x1: int, x2:int)
  ensures m == 0 ==>
    eval_linear(m, b, x1) == eval_linear(m, b, x2)
{
}


lemma zero_slope(m: int, b: int, x1: int, x2:int)
  requires m == 0
  ensures eval_linear(m, b, x1) == eval_linear(m, b,
x2)
{
}
```

Boolean operators

```
!
&&
||
==
==>
<==>
forall
exists
```

- Shorter operators have higher precedence
  `P(x) && Q(x) ==> R(S)`

- Bulleted conjunctions / disjunctions

```
(&& ( P(x))
 && ( Q(y))
 && ( R(x))==>(S(y))
 && ( T(x, y)))
```

- Parentheses are a good idea around
  forall, **exists**, **==>**

Quantifier syntax: forall

The type of **a** is typically inferred

```
forall a :: P(a)

forall a | Q(a) :: R(a)
```

expression forms

```
forall a | Q(a)
   ensures R(a)
{
}
```

statement form

VSCode transition

Quantifier syntax: exists

`forall's` evil twin

`exists a :: P(a)`

E.g. exists n:nat :: 2*n == 4

Dafny cannot prove exists without a witness

VSCode transition

if-then-else expressions

```
if a < b then P(a) else P(b)

      <==>

( a < b && P(a) ) || ( !(a < b) && P(b) )
```

<u>If-then-else expressions work with other types:</u>

```
if a < b then a + 1 else b - 3
```

Sets

```
a: set<int>, b: set<int>     set is a templated type
{1, 3, 5}   {}               set literals
7 in a                       element membership
a <= b                       subset
a + b                        union
a - b                        difference
a * b                        intersection
a == b                       equality (works with all mathematical objects)
|a|                          set cardinality
set x: nat |                 set comprehension
  x < 100 && x % 2 == 0
```

Sequences

`a: seq<int>, b: seq<int>`	seq is a templated type
`[1, 3, 5]    []`	sequence literal
`7 in a`	element membership
`a + b`	concatenation
`a == b`	equality *(works with all mathematical objects)*
`|a|`	sequence length
`a[2..5]      a[3..]`	sequence slice
`seq(5, i => i * 2)`	sequence comprehension

```
seq(5, i requires 0<=i
         => sqrt(i))
```

Maps

```
a: map<int, set<int>>
map[2:={2}, 6:={2,3}]
7 in a
7 in a.Keys
a == b
a[5 := {5}]
map k | k in Evens()
     :: k/2
```

- map is a templated type
- map literal
- key membership
- alternative form of key membership
- equality *(works with all mathematical objects)*
- map update *(not a mutation)*
- map comprehension

var is mathematical **let.**
It introduces an equivalent shorthand for another expression.

```
lemma foo()
{
  var set1 := { 1, 3, 5, 3 };
  var seq1 := [ 1, 3, 5, 3 ];

  assert forall i | i in set1 :: i in seq1;
  assert forall i | i in seq1 :: i in set1;
  assert |set1| < |seq1|;
}
```

Algebraic datatypes ("struct" and "union")

```
datatype HAlign = Left | Center | Right
```

new name we're defining

disjoint constructors

```
datatype VAlign = Top | Middle | Bottom

datatype TextAlign = TextAlign(hAlign:HAlign, vAlign:VAlign)
```

multiplicative constructor

```
datatype Order =   Pizza(toppings:set<Topping>)
                 | Shake(flavor:Fruit, whip: bool)
```

Hoare logic composition

```
lemma DoggiesAreQuadrupeds(pet: Pet)
  requires IsDog(pet)
  ensures |Legs(pet)| == 4 { … }

lemma StaticStability(pet: Pet)
  requires |Legs(pet)| >= 3
  ensures IsStaticallyStable(pet) { … }

lemma DoggiesAreStaticallyStable(pet: Pet)
  requires IsDog(pet)
  ensures IsStaticallyStable(pet)
{
  DoggiesAreQuadrupeds(pet);
  StaticStability(pet);
}
```

Lemmas can return results

```
lemma EulerianWalk(g: Graph) returns (p: Path)
  requires |NodesWithOddDegree(g)| <= 2
  ensures EulerWalk(g, p)
```

Detour to Imperativeland

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 <= x < |a|
  && (forall i :: 0 <= i < |a| ==> a[i] <= a[x])
}
```

Imperativeland

```
method findMaxIndex(a:seq<int>) returns (x:int)
  requires |a| > 0
  ensures IsMaxIndex(a, x)
{
  var i := 1;
  var ret := 0;
  while(i < |a|)
    invariant 0 <= i <= |a|
    invariant IsMaxIndex(a[..i], ret)
  {
    if(a[i] > a[ret]) {
      ret := i;
    }
    i := i + 1;
  }
  return ret;
}
```

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 <= x < |a|
  && (forall i :: 0 <= i < |a| ==> a[i] <=
a[x])
}
```