# Lecture 12
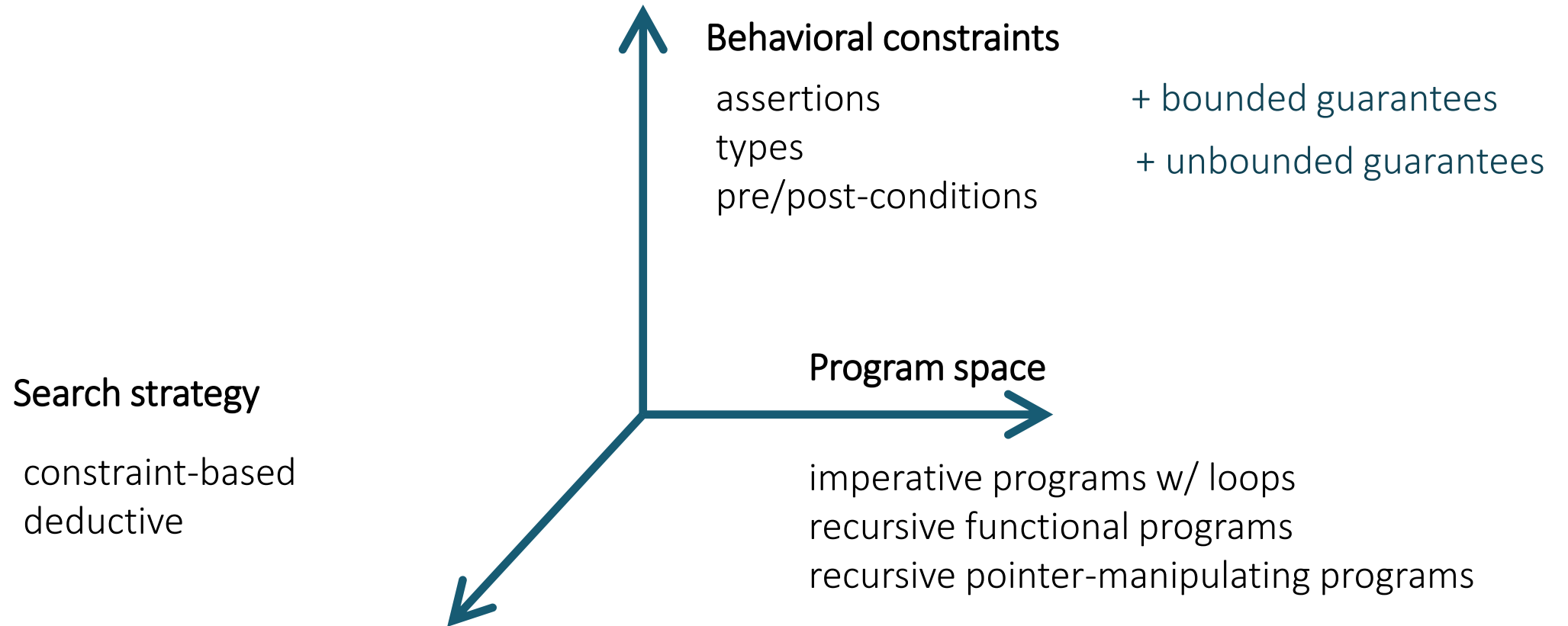# Hoare Logic

*Nadia Polikarpova*
*(some material from Peter Müller, ETH Zurich)*
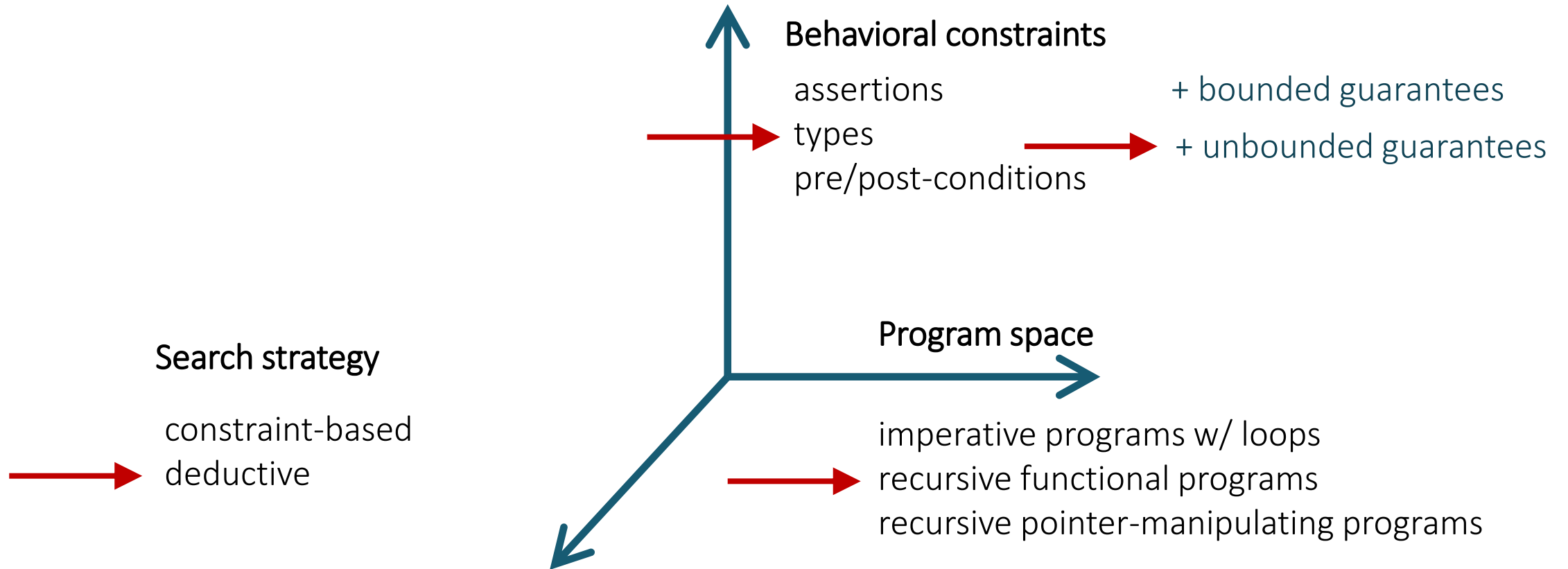
# Module II

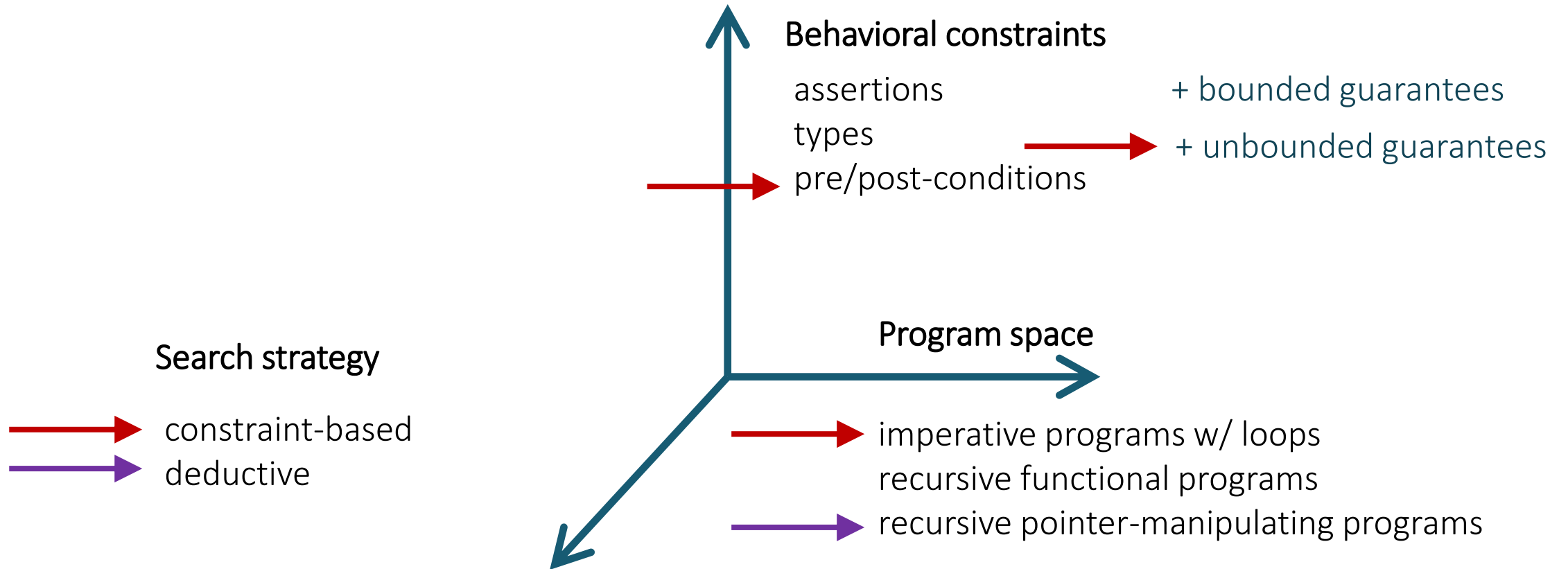Behavioral constraints

assertions          + bounded guarantees
types                 + unbounded guarantees
pre/post-conditions

Program space

Search strategy

constraint-based
deductive

imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

# Last week

**Behavioral constraints**

assertions                        + bounded guarantees
types                             + unbounded guarantees
pre/post-conditions

**Program space**

imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

**Search strategy**

constraint-based
deductive

# This week



Behavioral constraints

assertions          + bounded guarantees
types               + unbounded guarantees
pre/post-conditions

Program space

Search strategy

constraint-based    imperative programs w/ loops
deductive           recursive functional programs
                    recursive pointer-manipulating programs

# Constraint-based synthesis

Behavioral constraints
= assertions, reference
implementation, pre/post

Structural constraints

encoding

$$\exists c . \forall x . Q(c, x)$$

# Why is this hard?

```
Euclid (int a, int b) returns (int x)
    requires  a > 0 ∧ b > 0
    ensures  x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

infinitely many inputs

infinitely many paths!

# Loop unrolling

```
Euclid (int a, int b) returns (int x)
    requires a > 0 ∧ b > 0
    ensures   x = gcd(a, b)
{
    int x , y := a, b;
    while (x != y) {
        if (x > y) x := ??*x + ??*y + ??;
        else y := ??*x + ??*y + ??;
}}
```

Unroll with depth = 1

```
if (x != y) {
    if (x > y)
        x := ??*x + ??*y + ??;
    else
        y := ??*x + ??*y + ??;
    assert !(x != y);
}
```

# What's wrong with unrolling?

```
Euclid (int a, int b) returns (int x)
    requires a > 0 ∧ b > 0
    ensures  x = gcd(a, b)
{
    int x , y := a, b;
    while (x != y) {
        if (x > y) x := ??*x + ??*y + ??;
        else y := ??*x + ??*y + ??;
}}
```

Unroll with depth = 1

```
if (x != y) {
    if (x > y)
        x := ??*x + ??*y + ??;
    else
        y := ??*x + ??*y + ??;
    assert !(x != y);
}
```

Unsatisfiable sketch

# What's wrong with unrolling?

What if we restrict inputs to [1, 2]?

Unsound solution!

```
Euclid (int a, int b) returns (int x)
    requires a > 0 ∧ b > 0
    ensures   x = gcd(a, b)
{
    int x , y := a, b;
    while (x != y) {
        if (x > y) x := ??*x + ??*y + ??;
        else y := ??*x + ??*y + ??;
}}
```
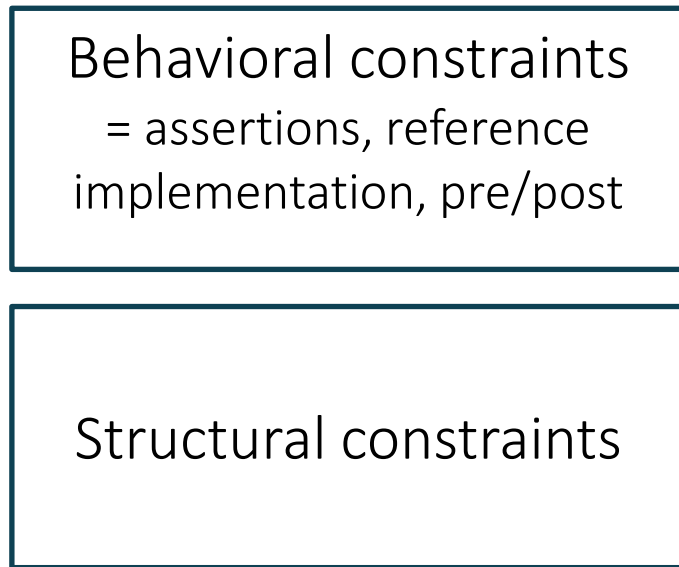
Unroll with depth = 1

```
if (x != y) {
    if (x > y)
        x := 0 *x + 0 *y + 1 ;
    else
        y := 0 *x + 0 *y + 1 ;
    assert !(x != y);
}
```

# Constraint-based synthesis

Behavioral constraints
= assertions, reference
implementation, pre/post

Structural constraints

encoding

$$\exists c . \forall x . Q(c, x)$$

If we want to synthesize programs that are correct on all inputs, we need a better way to deal with loops!

# Solution

Hoare logic = a program logic for simple imperative programs
- in particular: loop invariants

# The Imp language

```
e ::= n | x |
      e + e | e - e | e * e |
      e = e | e < e | !e | e && e
c ::= skip                    |
      x := e                  |
      c ; c                   |
      if e then c else c |
      while e do c
```

# Hoare triples

Properties of programs are specified as judgments

$$\{P\}\, c\, \{Q\}$$

where $c$ is a command and $P, Q : \sigma \to Bool$ are predicates

- e.g. if $\sigma = [x \mapsto 2]$ and $P \equiv x > 0$ then $P\,\sigma = \top$

Terminology

- Judgments of this kind are called *(Hoare) triples*
- $P$ is called precondition
- $Q$ is called postcondition

# Meaning of triples

The meaning of $\{P\}\ c\ \{Q\}$ is:

- **if** $P$ holds in the initial state $\sigma$, and
- **if** the execution of $c$ from $\sigma$ terminates in a state $\sigma'$
- **then** $Q$ holds in $\sigma'$

This interpretation is called *partial correctness*

- termination is not essential

Another possible interpretation: *total correctness*

- **if** $P$ holds in the initial state $\sigma$
- **then** the execution of $c$ from $\sigma$ terminates in a state (call it $\sigma'$)
- **and** $Q$ holds in $\sigma'$

# Example: swap

$\{\top\}$

    x := x + y; y := x - y; x := x - y

~~$\{x = y \land y = x\}$~~

We have to express that **y** in the final state is equal to **x** in the initial state!

# Logical variables

$\{x = N \land y = M\}$

    x  :=  x  +  y;  y  :=  x  -  y;  x  :=  x  -  y

$\{x = M \land y = N\}$

Assertions can contain *logical variables*
- may occur only in pre- and postconditions, not in programs
- the state maps logical variables to their values, just like normal variables

# Inference system

We formalize the semantics of a language by describing which judgments are valid about a program

An *inference system*

- a set of *axioms* and *inference rules* that describe how to derive a valid judgment

We combine axioms and inference rules to build *inference trees* (derivations)

# Semantics of skip

`skip` does not modify the state

$$\{\,P\,\}\,\texttt{skip}\,\{\,P\,\}$$

# Semantics of assignment

$$\{\,x > 0\,\} \quad \text{x} := \text{x} + 1 \quad \{\,???\,\}$$

$$\{\,???\,\} \quad \text{x} := \text{x} + 1 \quad \{\,x > 1\,\}$$

# Semantics of assignment

x := e assigns the value of e to variable x

$$\{ P[x \mapsto e] \} \quad x := e \quad \{ P \}$$

- Let $\sigma$ be the initial state
- Precondition: $(P[x \mapsto e])\sigma$, i.e., $P(\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])$
- Final state: $\sigma' = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$
- Consequently, $P$ holds in the final state

# Semantics of composition

Sequential composition `c1 ; c2` executes **c1** to produce an intermediate state and from there executes **c2**

$$\frac{\{P\} \; c_1 \; \{R\} \qquad \{R\} \; c_2 \; \{Q\}}{\{P\} \; c_1 ; c_2 \; \{Q\}}$$

# Example: swap

inference tree

leaves = axioms

$$\text{assign} \quad \frac{}{\{x = N + M \land y = N\} \ \texttt{x := x - y} \ \{x = M \land y = N\}}$$

$$\text{assign} \quad \frac{}{\{x = N + M \land y = M\} \ \texttt{y := x - y} \ \{x = N + M \land y = N\}}$$

edges = rules

$$\text{comp} \quad \frac{}{\{x = N + M \land y = M\} \ \texttt{y := x - y; x := x - y} \ \{x = M \land y = N\}}$$

$$\text{assign} \quad \frac{}{\{x = N \land y = M\} \ \texttt{x := x + y} \ \{x = N + M \land y = M\}}$$

$$\text{comp} \quad \frac{}{\{x = N \land y = M\} \ \texttt{x := x + y; y := x - y; x := x - y} \ \{x = M \land y = N\}}$$

root = triple to prove

# Proof outline

An alternative (more compact) representation of inference trees

$$\{x = N \wedge y = M\}$$
$$\Rightarrow$$
$$\{(x + y) - ((x + y) - y) = M \wedge (x + y) - y = N\}$$

```
x = x + y;
```

$$\{x - (x - y) = M \wedge x - y = N\}$$

```
y = x - y;
```

$$\{x - y = M \wedge y = N\}$$

```
x = x - y
```

$$\{x = M \wedge y = N\}$$

# Rule of consequence

$$\frac{\{P'\} \; c \; \{Q'\}}{\{P\} \; c \; \{Q\}} \quad \text{if} \quad P \Rightarrow P' \land Q' \Rightarrow Q$$

Corresponds to adding $\Rightarrow$ steps in a proof outline

Here $P \Rightarrow P'$ should be read as

- "We can prove for all states $\sigma$, that $P \; \sigma$ implies $P' \; \sigma$"

# Semantics of conditionals

$$\frac{\{P \land e\}\, c_1\, \{Q\} \qquad \{P \land \neg e\}\, c_2\, \{Q\}}{\{P\} \text{ if } e \text{ then } c_1 \text{ else } c_2\, \{Q\}}$$

# Example: absolute value

```
{⊤}
    if x < 0 then
        {⊤ ∧ x < 0}
    ⇒
        {−x ≥ 0}
        x := -x
        {x ≥ 0}
    else
        {¬(x < 0)}
    ⇒
        {x ≥ 0}
        skip
        {x ≥ 0}
{x ≥ 0}
```
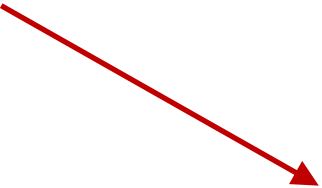
$$\frac{\{P \wedge e\}\, c_1\, \{Q\} \qquad \{P \wedge \neg e\}\, c_2\, \{Q\}}{\{P\}\ \text{if } e \text{ then } c_1 \text{ else } c_2\ \{Q\}}$$
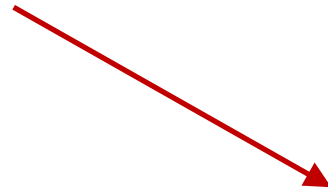
# Semantics of loops

We want to say:
- *P* holds initially
- after executing *c*
  - if *e* still holds, we execute it *c* again
  - otherwise, *Q* holds

$$\frac{\{?\}\ c\ \{?\}}{\{P\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{Q\}}$$

# Semantics of loops

loop invariant

$$\frac{\{I \wedge e\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{\neg e \wedge I\}}$$

# Example: GCD

$\{x = N \land y = M \land N > 0 \land M > 0\}$
  $\Rightarrow$
$\{I\}$
    **while** x != y **do**
       $\{I \land x \neq y\}$
        **if** x > y **then**
          x := x - y
        **else**
          y := y - x
       $\{I\}$
$\{I \land x = y\}$
  $\Rightarrow$
$\{x = \gcd(N, M)\}$

Guessing the loop invariant:

| x | y | N | M |
|---|---|---|---|
| 10 | 4 | 10 | 4 |
| 6 | 4 | 10 | 4 |
| 2 | 4 | 10 | 4 |
| 2 | 2 | 10 | 4 |

$$I \equiv \gcd(x, y) = \gcd(N, M)$$

# Example: GCD

$$\{x = N \land y = M \land N > 0 \land M > 0\}$$
$$\Rightarrow$$
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x, y > 0\}$$

```
while x != y do
```
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x, y > 0 \land x \neq y\}$$
```
    if x > y then
```
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x \neq y \land x > y\}$$
$$\Rightarrow$$
$$\{\text{gcd}(x - y, y) = \text{gcd}(N, M) \land x - y, y > 0\}$$
```
        x := x - y
```
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x, y > 0\}$$
```
    else
        y := y - x
```
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x, y > 0\}$$
$$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land x, y > 0 \land x = y\}$$
$$\Rightarrow$$
$$\{x = \text{gcd}(N, M)\}$$

# Termination

$$\frac{\{I \wedge e \wedge r = R\} \; c \; \{I \wedge r < R \wedge r \geq 0\}}{\{I\} \; \text{while } e \text{ do } c \; \{\neg e \wedge I\}}$$

# Example: GCD

```
while x != y do

    if x > y then
        x := x - y
    else
        y := y - x
```

# Example: GCD

$\{x = N \land y = M \land N > 0 \land M > 0\}$

$\Rightarrow$

$\{\gcd(x, y) = \gcd(N, M) \land x, y > 0\}$

   **while** x != y **do**

     $\{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x + y = R \land x \neq y\}$

      **if** x > y **then**

        x := x - y

      **else**

        y := y - x

     $\{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x + y < R \land x + y \geq 0\}$

$\{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x = y\}$

$\Rightarrow$

$\{x = \gcd(N, M)\}$

# Program Verification

```
method Euclid (a: int, b: int) returns (gcd: int)
  requires a > 0 && b > 0
  ensures x == gcd(a,b)
{
  var x, y := a, b;
  while (x != y)
    invariant y > 0 && x > 0 && gcd(x,y) == gcd(a,b)
    decreases x + y
  {
    if (x > y) {
      x := x - y;
    } else {
      y := y - x;
    }
  }
}
```

Dafny →

✅ correct!

❌ can't proof correctness

# Program synthesis

```
method Euclid (a: int, b: int) returns (gcd: int)
  requires a > 0 && b > 0
  ensures x == gcd(a,b)
{
  var x, y := ??;
  ??;
  while (??)
    invariant ??
    decreases ??
  {
   ??;
  }
  ??;
}
```

found a correct program!

```
var x, y := a, b;
while (x != y)
  invariant y > 0 && x > 0 && gcd(x,y) == gcd(a,b)
  decreases x + y
{
  if (x > y) {
    x := x - y;
  } else {
    y := y - x;
  }
}
```

can't find a (program, invariant) pair that I can prove correct

# Verification → synthesis

Srivastava, Gulwani, Foster: [From program verification to program synthesis](). POPL'10
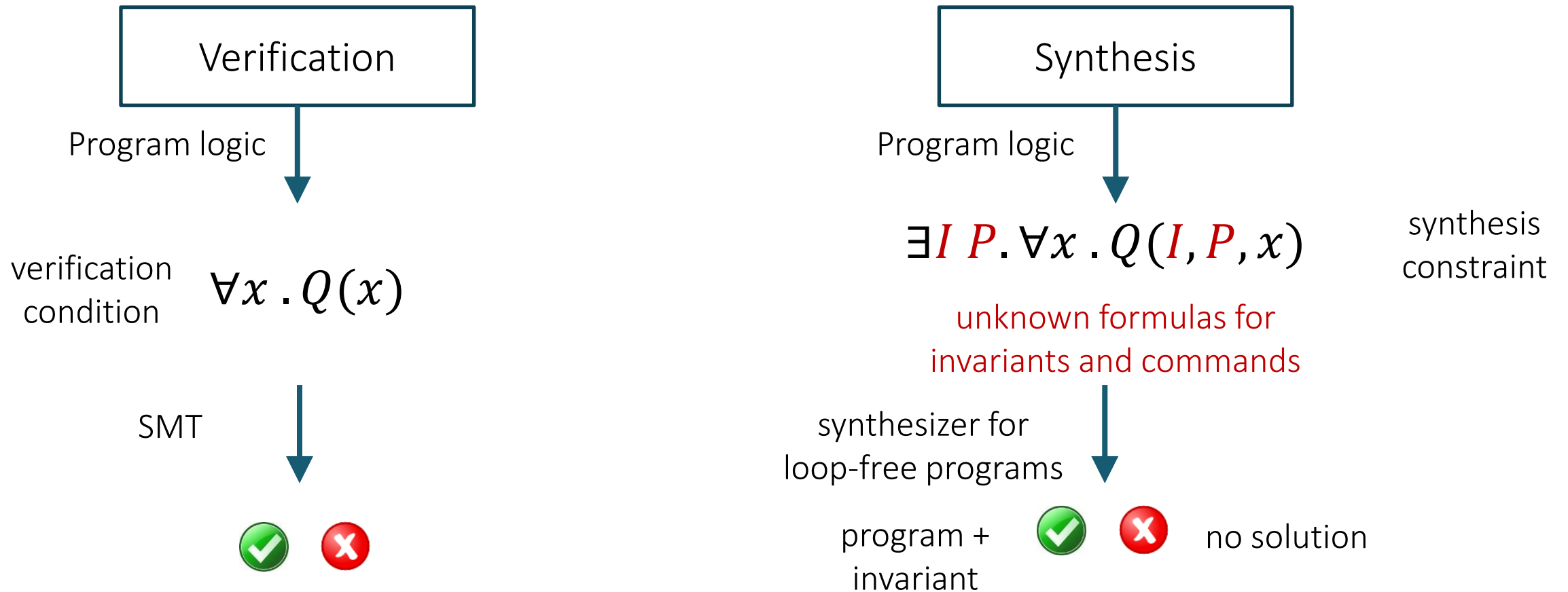
- idea: make constraint-based synthesis unbounded by synthesizing loop invariants alongside programs
- synthesized some looping programs with integers, including Bresenheim algorithm
- won "Most Influential Paper" at POPL'20!

Qiu, Solar-Lezama: [Natural Synthesis of Provably-Correct Data-Structure Manipulations](). OOPSLA'17

- same approach for pointer-manipulating programs

# Verification → synthesis

| Verification | Synthesis |
|---|---|

Program logic

Program logic

verification
condition

$$\forall x \, . \, Q(x)$$

synthesis
constraint

$$\exists I \, P \, . \, \forall x \, . \, Q(I, P, x)$$

unknown formulas for
invariants and commands

SMT

synthesizer for
loop-free programs

program +
invariant

no solution

# How verification works

Verification

$$\forall x . Q(x)$$

# Step 1: eliminate loops

```
{pre}
  init;
{inv }
```

```
{pre}
   init;
   while (c)
      invariant inv
   { body; }
   final;
{post}
```

```
{inv ∧ cond }
   body;
{inv}
```

```
{inv ∧ ¬(cond) }
   final;
{post}
```

# Step 2: generate VCs

```
{pre}
  init
  while (c)
    invariant inv;
  { body }
  final
{post}
```

$\{pre\}$
   init
$\{inv\}$

$\rightarrow$

$\{pre\}$
   $init(x, x')$
$\{inv[x \mapsto x']\}$

$\{inv \wedge c\}$
   body
$\{inv\}$

$\rightarrow$

$\{inv \wedge c\}$
   $body(x, x')$
$\{inv[x \mapsto x']\}$

$\{inv \wedge \neg c\}$
   final
$\{post\}$

$\rightarrow$

$\{inv \wedge \neg c\}$
   $final(x, x')$
$\{post\}$

$$\forall x \, .$$
$$pre \wedge init \Rightarrow inv'$$
$$\wedge$$
$$inv \wedge c \wedge body \Rightarrow inv'$$
$$\wedge$$
$$inv \wedge \neg c \wedge final \Rightarrow post$$

verification condition

$$\forall x \, . \, Q(x)$$
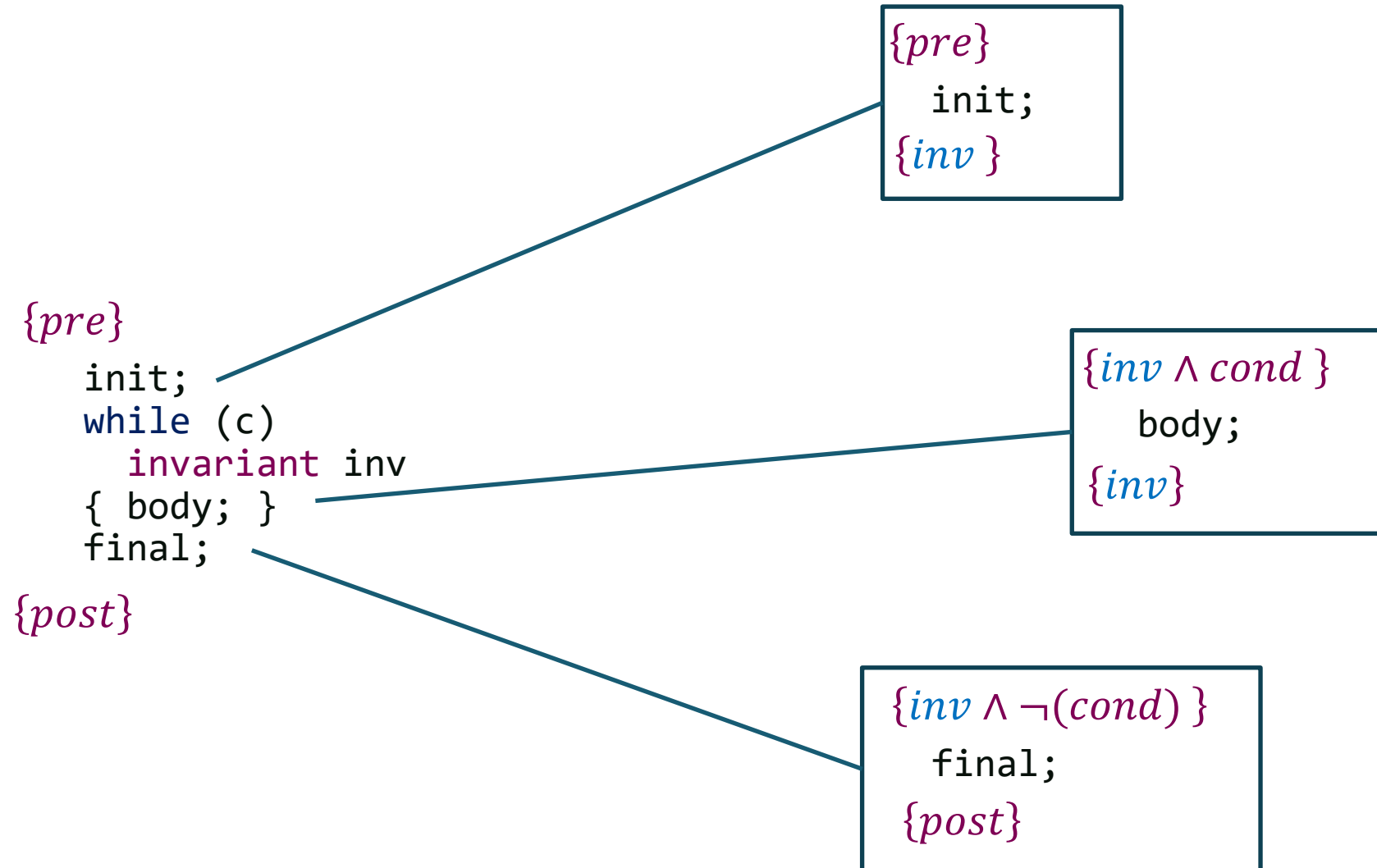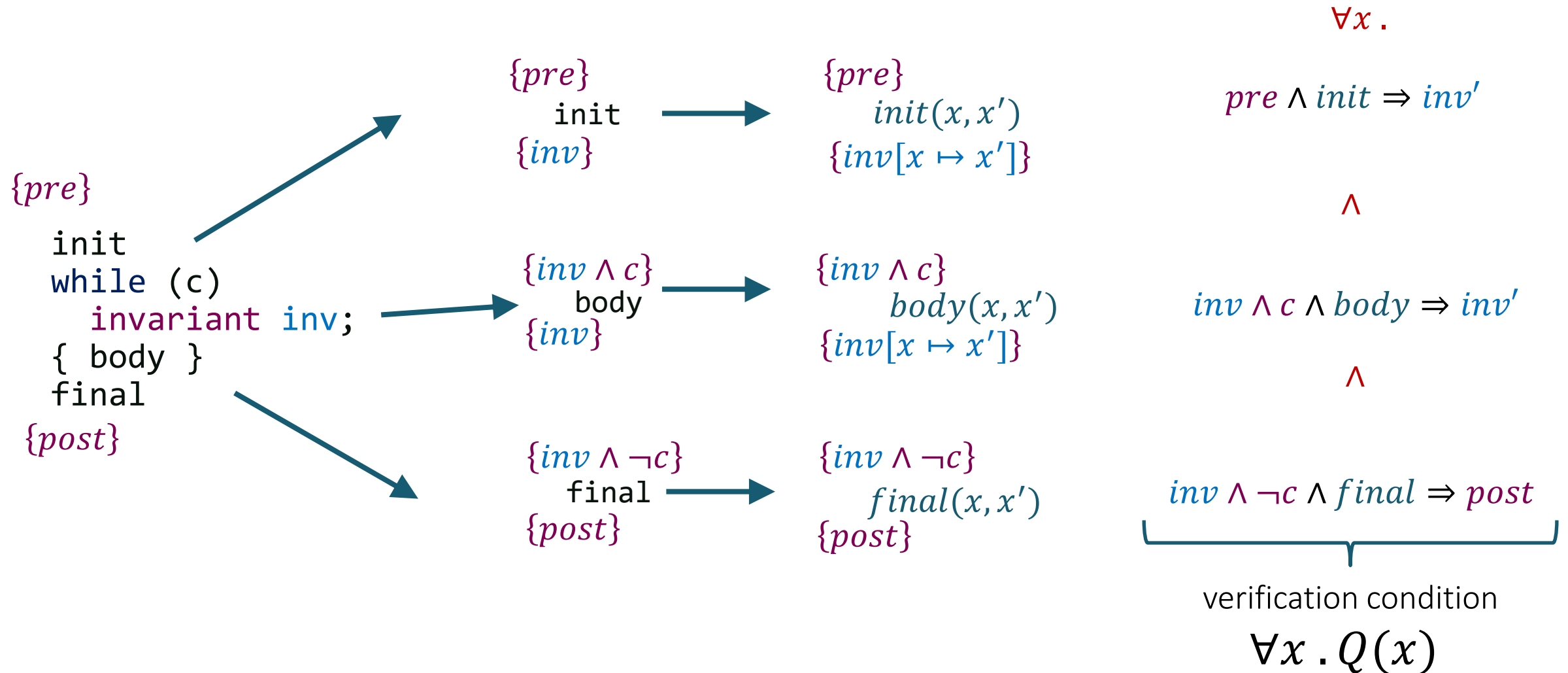
# From verification to synthesis

Verification

$$\forall x . Q(x)$$

$$\equiv$$

$$\exists x . \neg Q(x)$$

SMT

UNSAT / SAT

Synthesis

$$\exists I \ P . \forall x . Q(I, P, x)$$

# Program synthesis

```
{pre}

  ??
  while (??)
    invariant ??;
  { ?? }
  ??
{post}
```

$\{pre\}$
$\quad S_i(x, x')$
$\{I[x \mapsto x']\}$

$\{I \wedge G\}$
$\quad S_b(x, x')$
$\{I[x \mapsto x']\}$

$\{I \wedge \neg G_0\}$
$\quad S_f(x, x')$
$\{post\}$
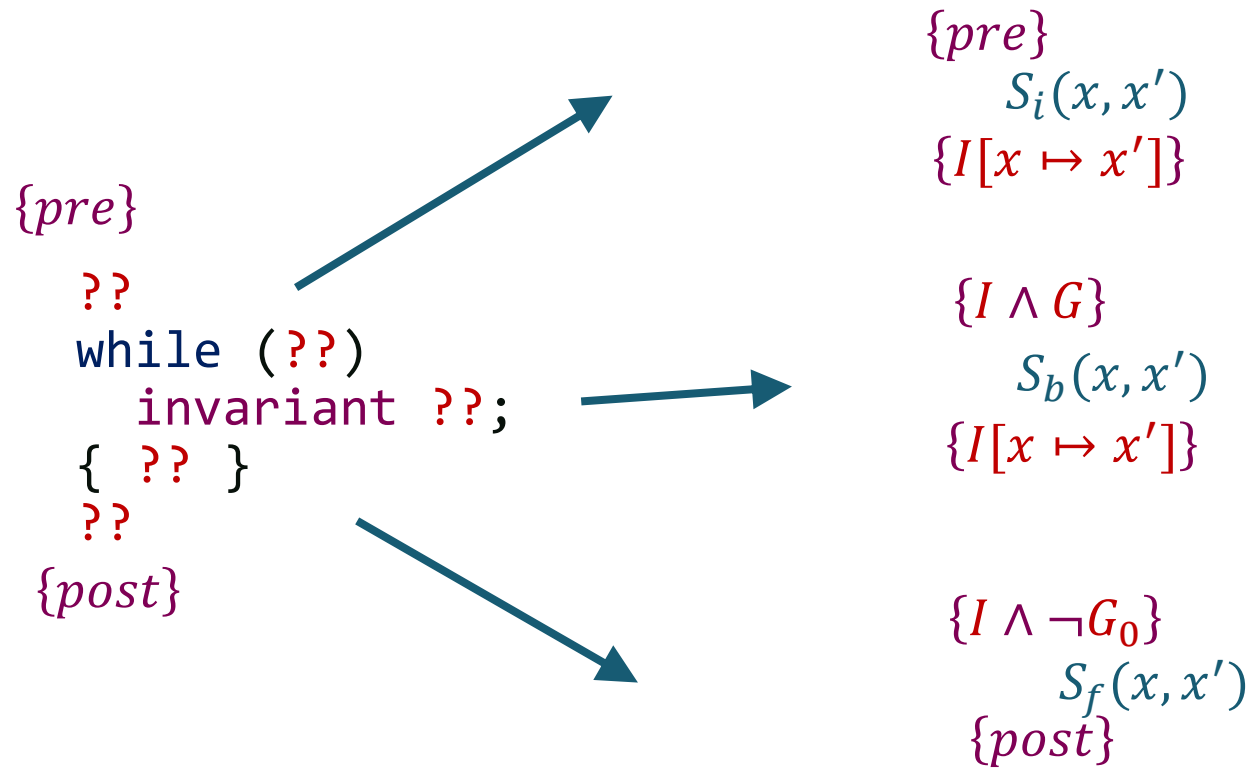
$\exists S\ G\ I. \forall x\ .$

$pre \wedge S_i \Rightarrow I'$

$\wedge$

$I \wedge G \wedge S_b \Rightarrow I'$

$\wedge$

$I \wedge \neg G \wedge S_f \Rightarrow post$

synthesis constraint

$\exists I\ P. \forall x\ . Q(I, P, x)$

# Synthesis constraints

$$pre \wedge S_i \Rightarrow I'$$

$$I \wedge G \wedge S_b \Rightarrow I'$$

$$I \wedge \neg G \wedge S_f \Rightarrow post$$

Domain for $I, G$: formulas over program variables

Domain for $S = \{x' = e_x \wedge y' = e_y \wedge \cdots \mid e_x, e_y, \ldots \in Expr\}$

- conjunction of equalities, one per variables

# Solving synthesis constraints

$$pre \wedge S_i \Rightarrow I'$$

$$I \wedge G \wedge S_b \Rightarrow I'$$

$$I \wedge \neg G \wedge S_f \Rightarrow post$$

Can be solved this with...

- SyGuS solvers
- Sketch
  - Look we made an unbounded synthesizer out of Sketch!
- VS3 uses Lattice search
  - More efficient for predicates