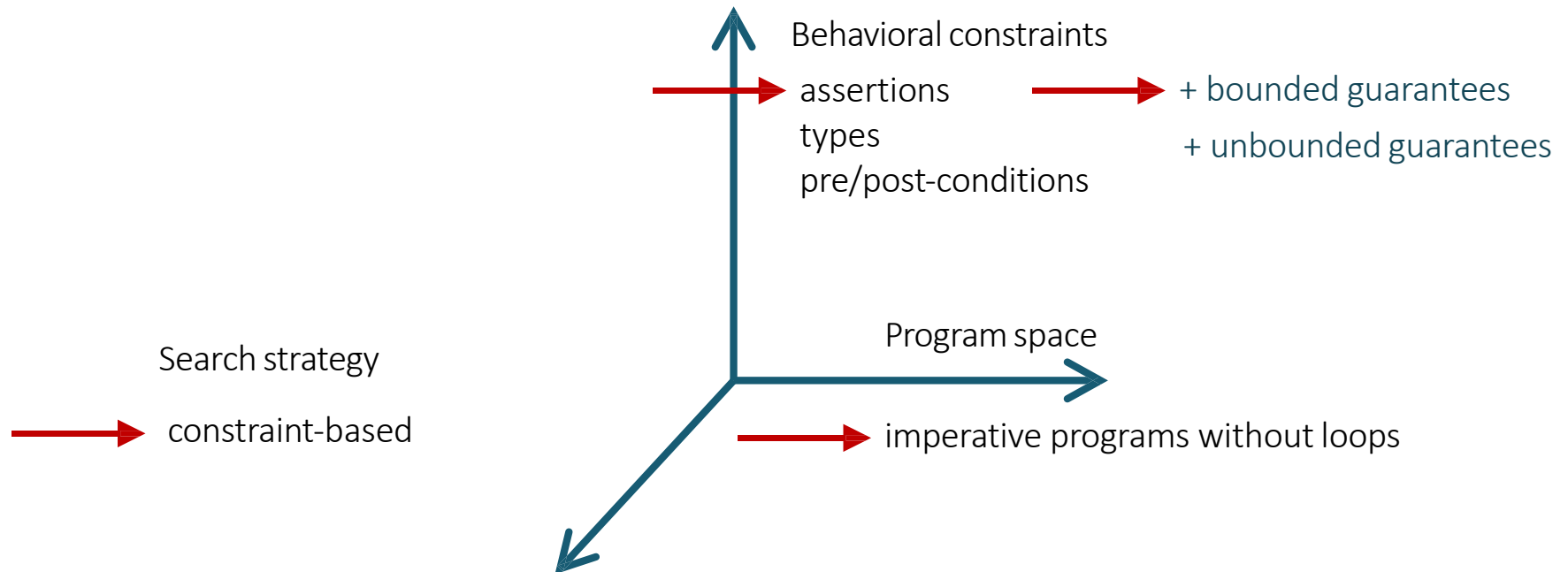


# **Lecture 11**

## **Hoare Logic**

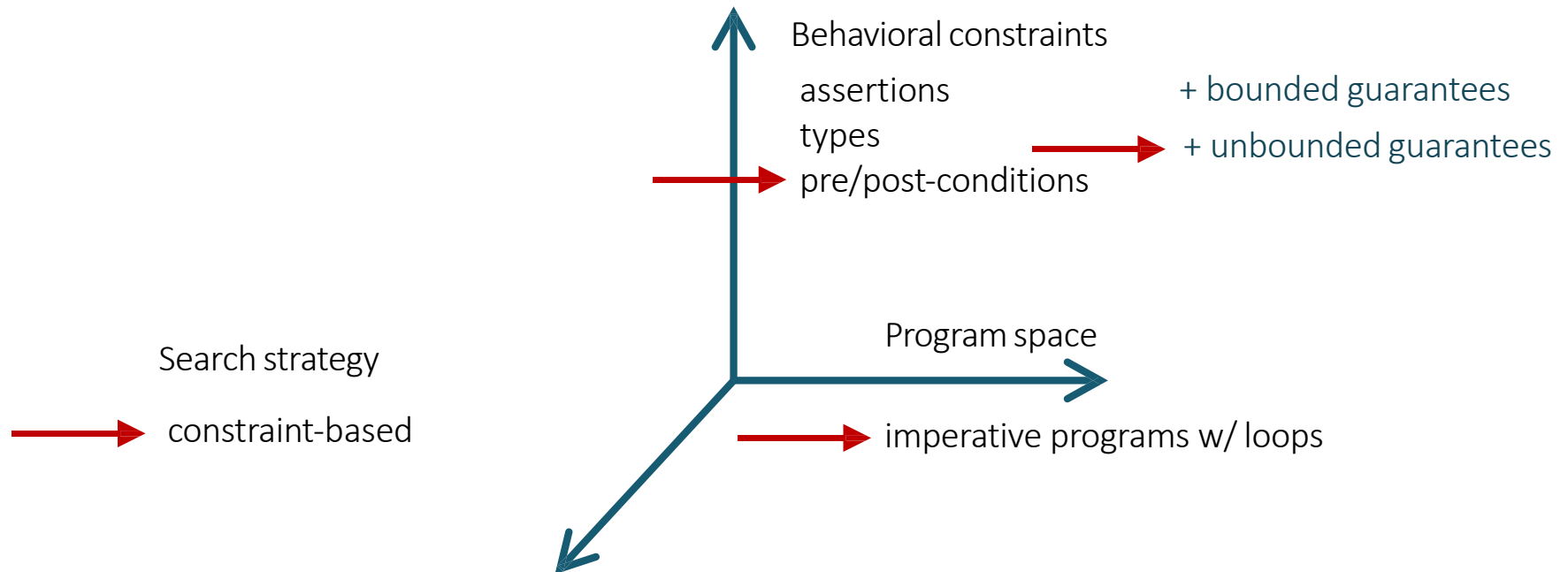
# This week

---



# This week

---



# Why is this hard?

---

```
Euclid (int a, int b) returns (int x)
  requires  $a > 0 \wedge b > 0$ 
  ensures  $x = \text{gcd}(a, b)$ 
{
  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**  $a > 0 \wedge b > 0$

**ensures**  $x = \text{gcd}(a, b)$

```
{  
  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**  $a > 0 \wedge b > 0$

**ensures**  $x = \text{gcd}(a, b)$

{

**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 inputs are 2-bit words?

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 els c  
      while e do c e
```

# Hoare triples

---

Properties of programs are specified as judgments

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

where  $c$  is a command and  $P, Q: \sigma \rightarrow \text{Bool}$  are predicates

- e.g. if  $\sigma = [x \mapsto 2]$  and  $P \equiv x > 0$  then  $P \sigma = \text{T}$

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

---

$\{T\}$

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

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

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

# Logical variables

---

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

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

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

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

$$\{ P \} \text{ skip } \{ P \}$$



# Semantics of assignment

---

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

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

- 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  $c_1 ; c_2$  executes  $c_1$  to produce an intermediate state and from there executes  $c_2$

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

# Example: swap

inference tree

leaves = axioms

assign  $\frac{}{\{x = N + M \wedge y = N\} \quad x := x - y \quad \{x = M \wedge y = N\}}$

assign  $\frac{}{\{x = N + M \wedge y = M\} \quad y := x - y \quad \{x = N + M \wedge y = N\}}$

edges = rules

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

assign  $\frac{}{\{x = N \wedge y = M\} \quad x := x + y \quad \{x = N + M \wedge y = M\}}$

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

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

---

$$\frac{\{P'\} c \{Q'\}}{\{P\} c \{Q\}} \text{ if } P \Rightarrow P' \wedge Q' \Rightarrow Q$$

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

Here  $R \Rightarrow S$  should be read as

- “We can prove for all states  $\sigma$ , that  $R \sigma$  implies  $S \sigma$ ”

# Semantics of conditionals

---

$$\frac{\{P \wedge e\} c_1 \{Q\} \quad \{P \wedge \neg e\} c_2 \{Q\}}{\{P\} \text{ if } e \text{ then } c_1 \text{ else } c_2 \{Q\}}$$

# Example: absolute value

---

```
{T}  
  if x < 0 then  
    {x < 0}  
    ⇒ {-x ≥ 0}  
    x := -x  
    {x ≥ 0}  
  else  
    ⇒ {¬(x < 0)}  
    {x ≥ 0}  
    skip  
    {x ≥ 0}  
{x ≥ 0}
```

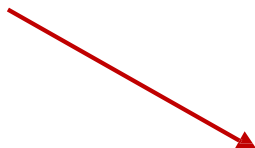
$$\frac{\{P \wedge e\} c_1 \{Q\} \quad \{P \wedge \neg e\} c_2 \{Q\}}{\{P\} \text{ if } e \text{ then } c_1 \text{ else } c_2 \{Q\}}$$

# 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

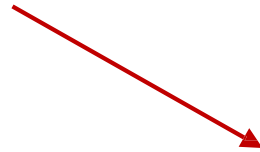

$$\frac{\{?\} c \{?\}}{\{P\} \text{ while } e \text{ do } c \{Q\}}$$



# Semantics of loops

---

loop invariant


$$\{I \wedge e\} c \{I\}$$

---

$$\{I\} \text{ while } e \text{ do } c \{ \neg e \wedge I \}$$

# Example: GCD

---

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

$\Rightarrow$

$\{I\}$

**while**  $x \neq y$  **do**

$\{I \wedge x \neq y\}$

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

$x := x - y$

**else**

$y := y - x$

$\{I\}$

$\{I \wedge x = y\}$

$\Rightarrow$

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

Guessing the loop invariant:

x	y
10	4
6	4
2	4
2	2

$I \equiv \text{gcd}(x, y) = \text{gcd}(N, M) \wedge x, y > 0$

# Example: GCD


---

```
{x = N ∧ y = M ∧ N > 0 ∧ M > 0 }  
⇒  
{gcd(x, y) = gcd(N, M) ∧ x, y > 0}  
  while x != y do  
    {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x ≠ y }  
    if x > y then  
      {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x ≠ y ∧ x > y}  
      ⇒  
      {gcd(x - y, y) = gcd(N, M) ∧ x - y, y > 0}  
      x := x - y  
      {gcd(x, y) = gcd(N, M) ∧ x, y > 0}  
    else  
      y := y - x  
      {gcd(x, y) = gcd(N, M) ∧ x, y > 0}  
  {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x = y }  
  ⇒  
{x = gcd(N, M) }
```

# Termination

---

loop variant / ranking function /  
termination metric


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

# Example: GCD

---

```
while x != y do  
    if x > y then  
        x := x - y  
    else  
        y := y - x
```

# Example: GCD

---

```
{x = N ∧ y = M ∧ N > 0 ∧ M > 0 }  
⇒  
{gcd(x, y) = gcd(N, M) ∧ x, y > 0}  
  while x != y do  
    {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x + y = R ∧ x ≠ y}  
    if x > y then  
      x := x - y  
    else  
      y := y - x  
    {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x + y < R ∧ x + y ≥ 0 }  
  {gcd(x, y) = gcd(N, M) ∧ x, y > 0 ∧ x = y }  
  ⇒  
  {x = gcd(N, M) }
```

# Program verifiers

---

Dafny demo

<https://rise4fun.com/Dafny/29sh>

# 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;
    }
  }
}
```



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](#)

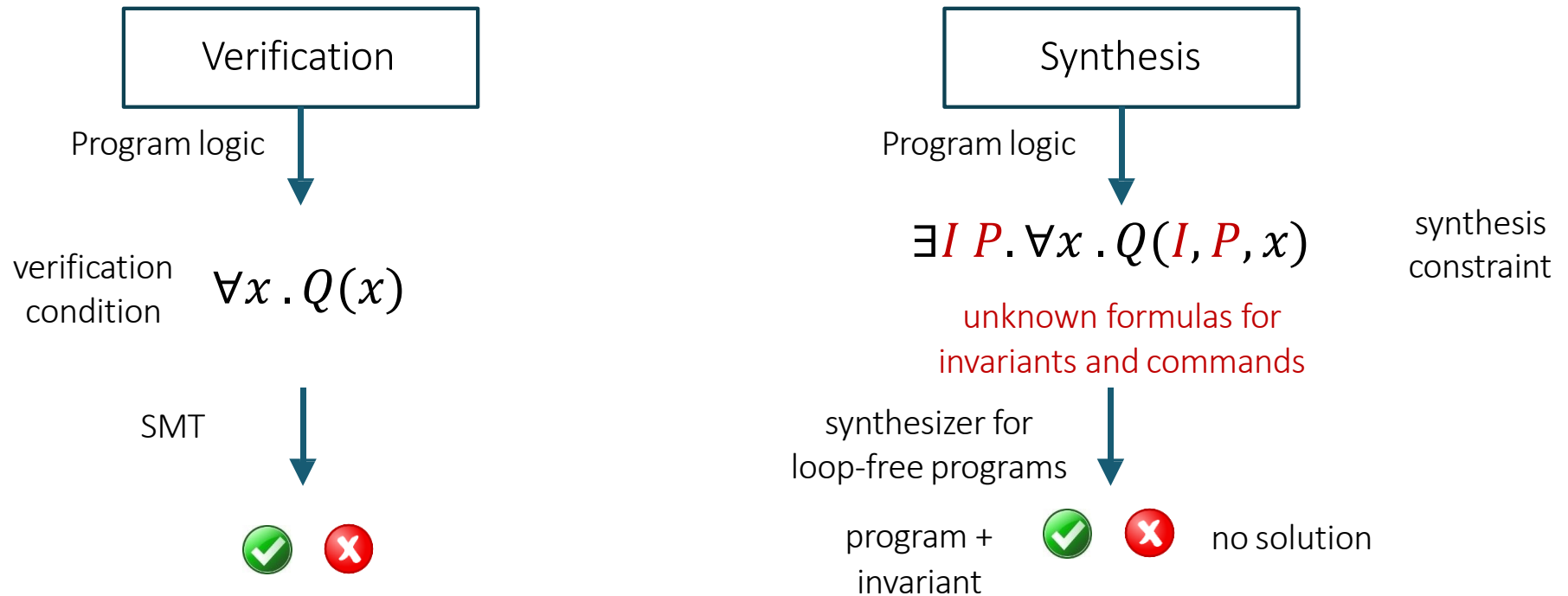
- idea: make constraint-based synthesis unbounded by synthesizing loop invariants alongside programs
- synthesized some looping programs with integers, including Bresenham algorithm
- won “Most Influential Paper” at POPL'20!

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

- same approach for pointer-manipulating programs

# Verification → Synthesis

---



# How verification works

---

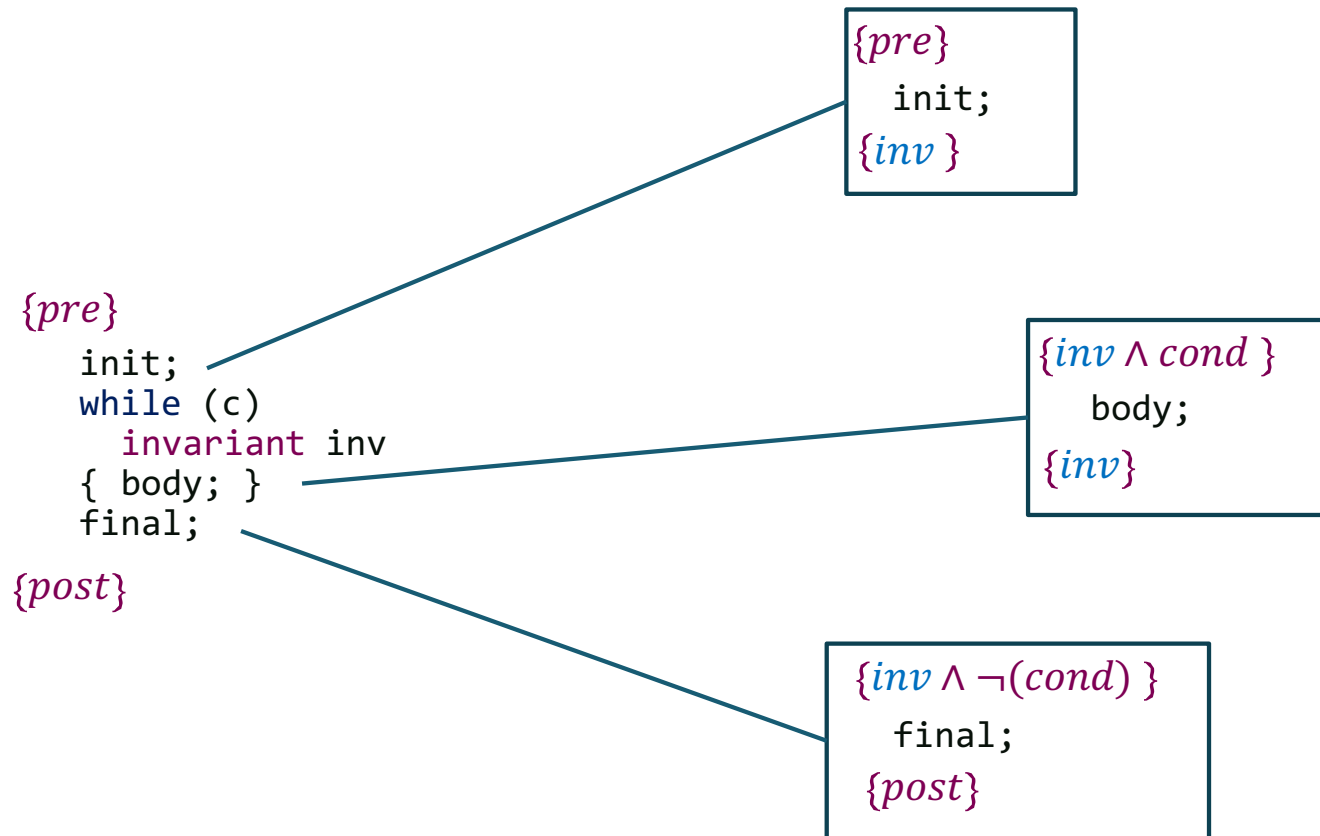
Verification



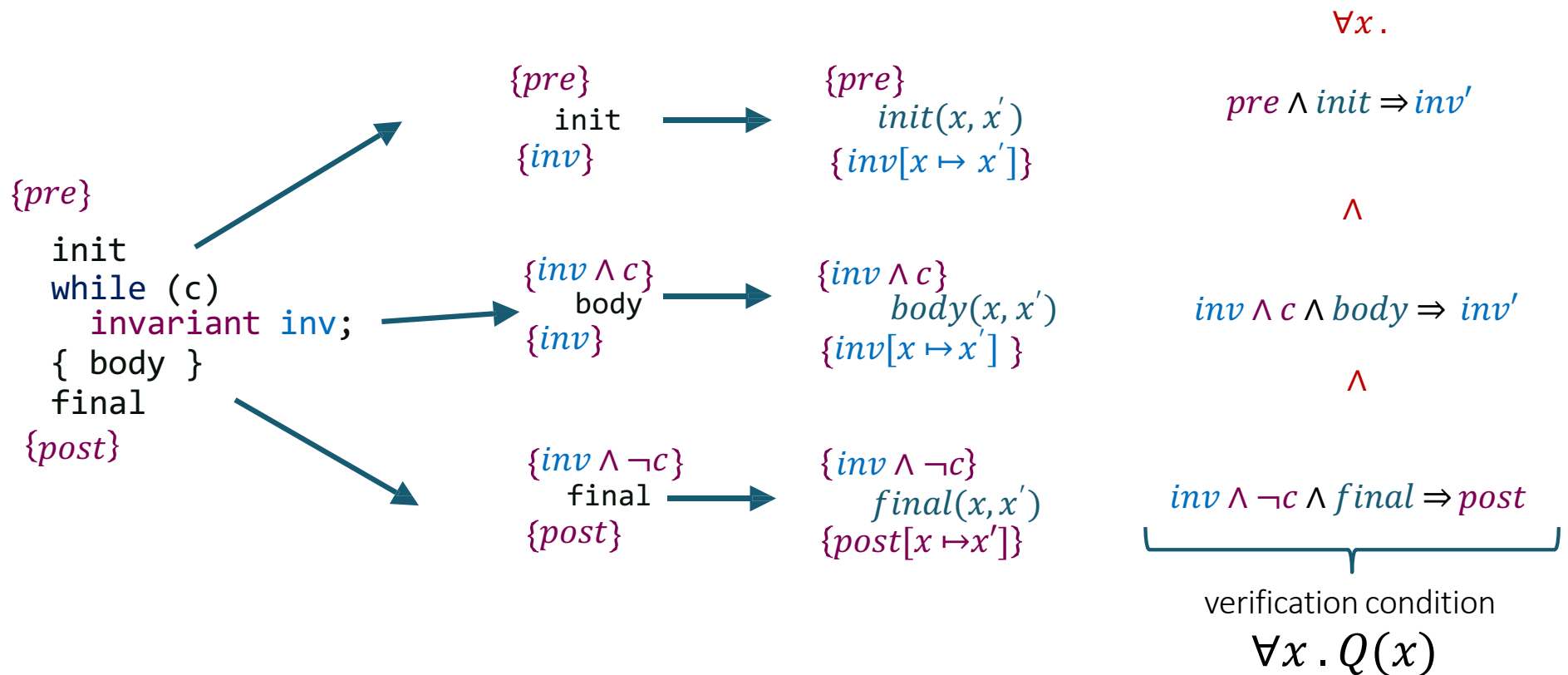
$\forall x . Q(x)$

# Step 1: eliminate loops

---



## Step 2: generate VCs



# From verification to synthesis

---

Verification

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

$$\stackrel{=}{\exists x . \neg Q(x)}$$

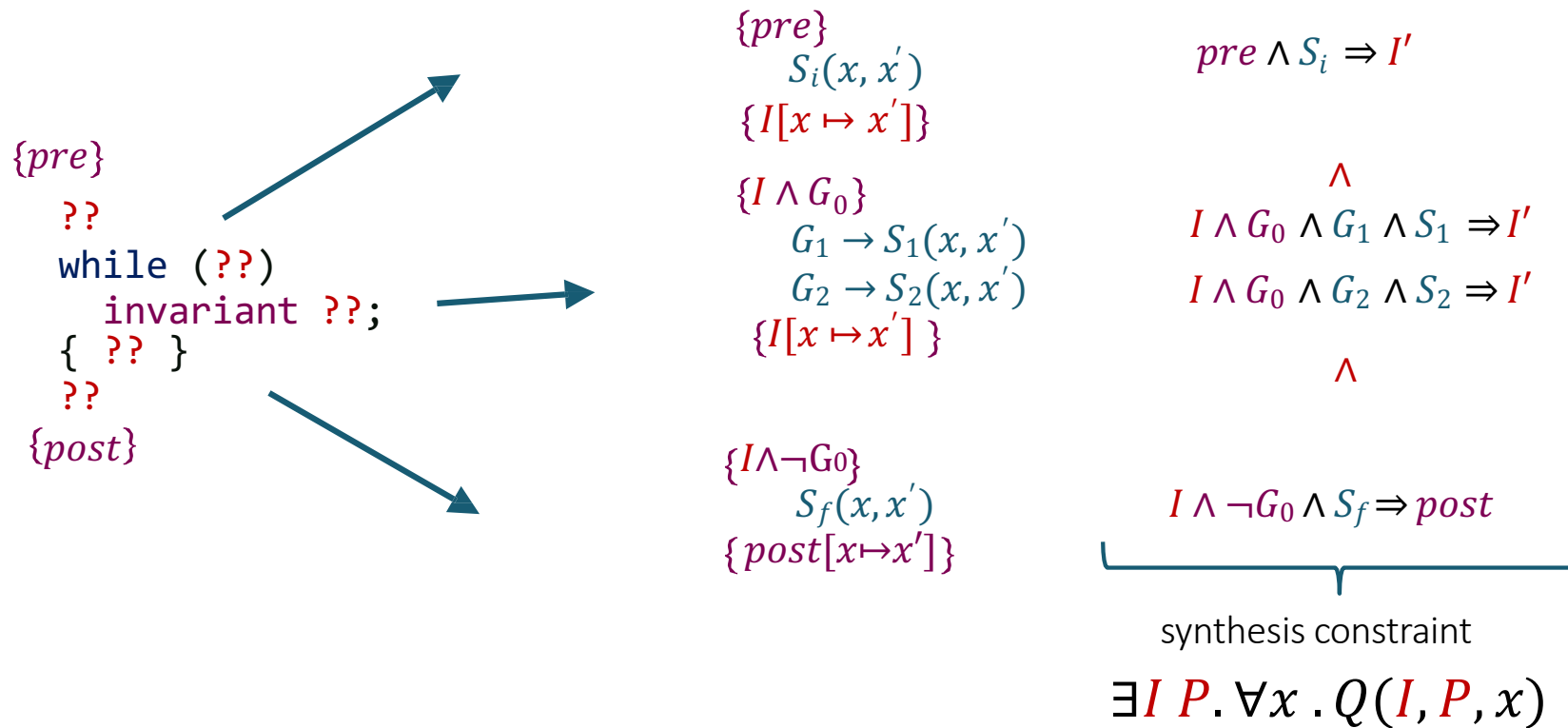
SMT

UNSAT / SAT

Synthesis

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

# Program synthesis





# Synthesis constraints

---

$$I \wedge G_i \wedge S_i \wedge \psi \Rightarrow I'$$

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

Domain for  $S_i = \{x' = e_x \wedge y' = e_y \wedge \dots \mid e_x, e_y, \dots \in Expr\}$

- conjunction of equalities, one per variables

# Solving synthesis constraints

---

$$I \wedge G_i \wedge S_i \wedge \psi \Rightarrow I'$$

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