EECS 598-008 & EECS 498-008: Intelligent Programming Systems

Lecture 11

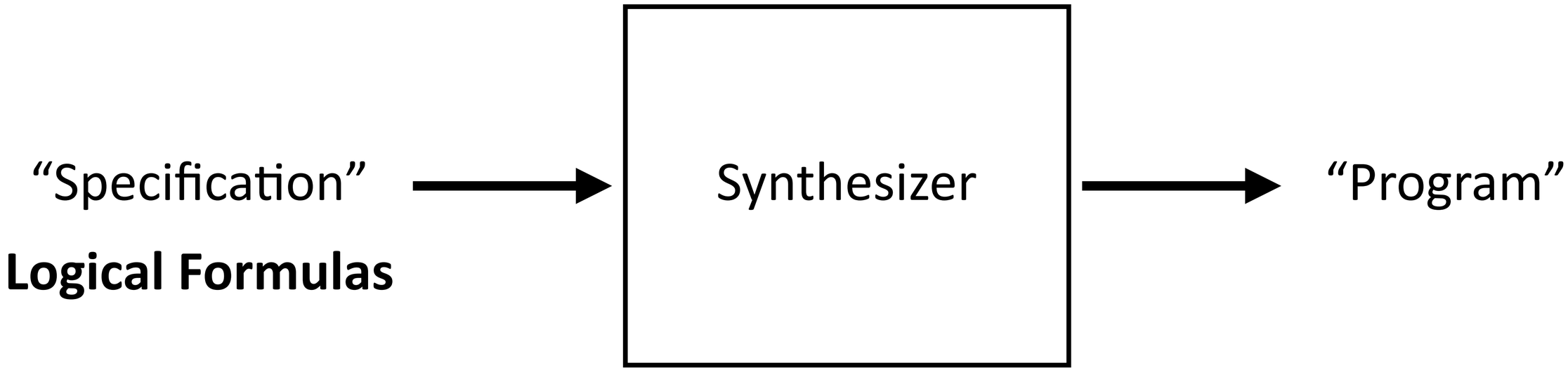
Announcements

- No lecture on Thursday (office hour instead)
- A3 due **midnight Wednesday (tomorrow)**
- First presentation Tuesday Oct 12
  - First paper review due **noon Monday Oct 11**
- Presentations
  - Speakers try to be in-person
- Friday discussion section
  - Discuss presentations with speakers in coming week
  - Schedule in advance if speakers want to discuss
  - Other students do not need to attend

So Far..

- Inductive program synthesis
  - Based on syntax-guided synthesis
  - Search space: DSL
  - Specification: examples
  - Search technique: top-down/bottom-up
  - Pruning: SMT-based deduction/equivalence class reduction
  - Prioritization: ranking/statistical models
  - Generalization: ranking/interaction/multi-modality

Today: Logical Specifications



Why Logical Formulas?

- Inductive specifications are incomplete

Why Logical Formulas?

- Inductive specifications are incomplete

	Benchmark	*Spec*	*SpecS*	*Time*	*TimeS*
LEON	strict sorted list delete	14	8	15.1	0.10
	strict sorted list insert	14	8	14.1	0.18
	merge sort	9	11	14.3	2.1
JEN	BST find	51	6	64.8	0.09
	bin. heap 1-element	80	5	61.6	0.02
	bin. heap find	76	6	51.9	0.38
MYTH	sorted list insert	12	8	0.12	0.25
	list rm adjacent dupl.	13	5	0.07	1.33
	BST insert	20	8	0.37	0.91
λ^2	list remove duplicates	7	13	231	0.36
	list drop	6	11	316.4	0.1
	tree find	12	6	4.7	0.29
ESC	list rm adjacent dupl.	n/a	5	1	1.33
	tree create balanced	n/a	7	0.24	0.14
	list duplicate each	n/a	7	0.16	0.05
MYTH2	BST insert	n/a	8	1.81	0.91
	sorted list insert	n/a	8	1.02	0.25
	tree count nodes	n/a	4	0.45	0.20

Too many examples

Why Logical Formulas?

- Inductive specifications are incomplete

	Benchmark	*Spec*	*SpecS*	*Time*	*TimeS*
LEON	strict sorted list delete	14	8	15.1	0.10
	strict sorted list insert	14	8	14.1	0.18
	merge sort	9	11	14.3	2.1
JEN	BST find	51	6	64.8	0.09
	bin. heap 1-element	80	5	61.6	0.02
	bin. heap find	76	6	51.9	0.38
MYTH	sorted list insert	12	8	0.12	0.25
	list rm adjacent dupl.	13	5	0.07	1.33
	BST insert	20	8	0.37	0.91
λ^2	list remove duplicates	7	13	231	0.36
	list drop	6	11	316.4	0.1
	tree find	12	6	4.7	0.29
ESC	list rm adjacent dupl.	n/a	5	1	1.33
	tree create balanced	n/a	7	0.24	0.14
	list duplicate each	n/a	7	0.16	0.05
MYTH2	BST insert	n/a	8	1.81	0.91
	sorted list insert	n/a	8	1.02	0.25
	tree count nodes	n/a	4	0.45	0.20

Too many examples

… which is fundamentally a problem for any inductive synthesis tool

Why Logical Formulas?

- Inductive specifications are incomplete
  - May need too many examples in some domains
  - Good examples are not always easy to provide
  - For safety critical domains, inductive specifications (alone) may not be a good idea

Why Logical Formulas?

- Inductive specifications are incomplete
  - May need too many examples in some domains
  - Good examples are not always easy to provide
  - For safety critical domains, inductive specifications (alone) may not be a good idea
- Use complete specifications
  - To fully characterize function behavior

Pre and Postconditions

- Specify functional correctness of programs
  - Precondition: a predicate that all valid inputs must satisfy
  - Postcondition: a predicate that all outputs must satisfy

Pre and Postconditions

- Specify functional correctness of programs
  - Precondition: a predicate that all valid inputs must satisfy
  - Postcondition: a predicate that all outputs must satisfy
  - If P executes from any state satisfying pre, it ends up in a state satisfying post
  - Complete specification

Pre and Postconditions

- Specify functional correctness of programs
  - Precondition: a predicate that all valid inputs must satisfy
  - Postcondition: a predicate that all outputs must satisfy
  - If P executes from any state satisfying pre, it ends up in a state satisfying post
  - Complete specification

- Pre is a promise made by the environment
  - Whenever P is called, its parameters satisfy precondition

Pre and Postconditions

- Specify functional correctness of programs
  - Precondition: a predicate that all valid inputs must satisfy
  - Postcondition: a predicate that all outputs must satisfy
  - If P executes from any state satisfying pre, it ends up in a state satisfying post
  - Complete specification

- Pre is a promise made by the environment
  - Whenever P is called, its parameters satisfy precondition
- Post is a promise made by the program, assuming precondition holds
  - After P terminates, its output satisfies the postcondition

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

- Postcondition: all outputs must satisfy

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

- Postcondition: all outputs must satisfy
  - Use *out* to denote output of max(x, y)

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

- Postcondition: all outputs must satisfy
  - Use *out* to denote output of max(x, y)
  - $out \geq x \wedge out \geq y$?

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

- Postcondition: all outputs must satisfy
  - Use *out* to denote output of max(x, y)
  - $out \geq x \wedge out \geq y$? Correct, but not a full specification

Pre and Postconditions

- max(x, y) returns max of two positive integers x, y

- Precondition: all valid inputs must satisfy
  - $x > 0 \wedge y > 0$

- Postcondition: all outputs must satisfy
  - Use *out* to denote output of max(x, y)
  - $out \geq x \wedge out \geq y$? Correct, but not a full specification
  - $(out = x \vee out = y) \wedge out \geq x \wedge out \geq y$?

Pre and Postconditions

- sort(arr) sorts int array in descending order

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - *True*

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output
  - $\left(\forall i\,.\;0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$?

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output
  - $\left(\forall i\,.\;0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$?
    - Not a full spec. Consider input [5,4,3,2,1] and output [0,0,0,0,0]

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output
  - $\left(\forall i\,.\,0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$

    $\wedge \left(\forall i\,.\,(\ 0 \leq i < len(out) \rightarrow \exists j\,.\,arr[j] = out[i]\ )\right)$?

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output
  - $\left(\forall i\,.\,0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$

    $\wedge \left(\forall i\,.\,(\ 0 \leq i < len(out) \rightarrow \exists j\,.\,arr[j] = out[i]\ )\right)$?
    - Not a full spec. Consider input [5,4,3,2,1]and output [1,1,1,1,1]

Pre and Postconditions

- sort(arr) sorts int array in descending order

- Precondition: all valid inputs must satisfy
  - $True$

- Postcondition: all outputs must satisfy
  - out is output
  - $\left(\forall i\,.\,0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$

$\wedge \left(\exists p : int \rightarrow int\,.\forall i.0 \leq i < len(out) \rightarrow \left(\exists j.0 \leq j < len(arr)\,.\; p(j) = i\right) \wedge out[i] = arr[p(i)]\right)$

Pre and Postconditions

$$\left(\forall i\,.\;0 \leq i < len(arr) - 1 \rightarrow out[i] \geq out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$$

$$\wedge\, \exists p : int \rightarrow int\,.\,\forall i\,.\left(0 \leq i < len(out) \rightarrow \left(\underline{\exists j.0 \leq j < len(arr)\,.\,p(j) = i} \wedge \underline{out[i] = arr[p(i)]}\right)\right)$$

Pre and Postconditions

$$\left(\forall i\,.\,0 \le i < len(arr) - 1 \rightarrow out[i] \ge out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$$

$$\wedge \exists p : int \rightarrow int\,.\,\forall i\,.\left(0 \le i < len(out) \rightarrow \left(\underline{\exists j.0 \le j < len(arr)\,.\,p(j) = i} \wedge \underline{out[i] = arr[p(i)]}\right)\right)$$

Essentially, out is a permutation of arr

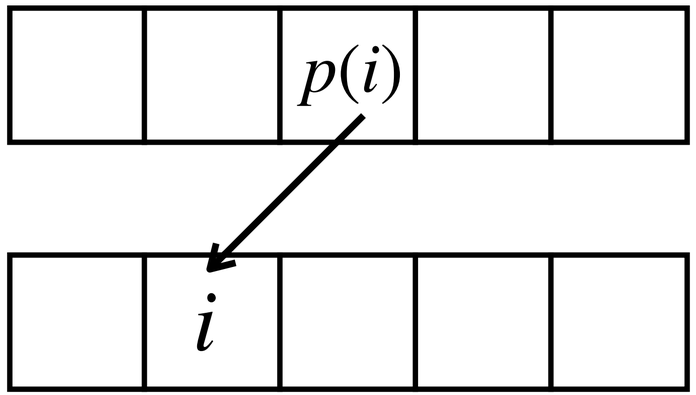
- p maps each element in out to some element in arr
- I.e., i-th element in out maps to $p(i)$-th element in arr

Pre and Postconditions

$$\left(\forall i\,.\,0 \le i < len(arr) - 1 \rightarrow out[i] \ge out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$$

$$\wedge\, \exists p : int \rightarrow int\,.\,\forall i\,.\left(0 \le i < len(out) \rightarrow \left(\underline{\exists j.0 \le j < len(arr)\,.\,p(j) = i} \wedge \underline{out[i] = arr[p(i)]}\right)\right)$$

Every element in the output is mapped from some element in input

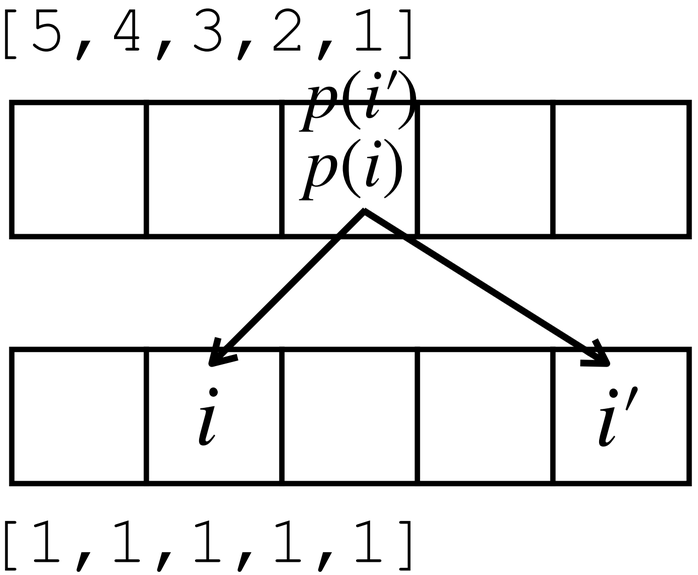


Pre and Postconditions

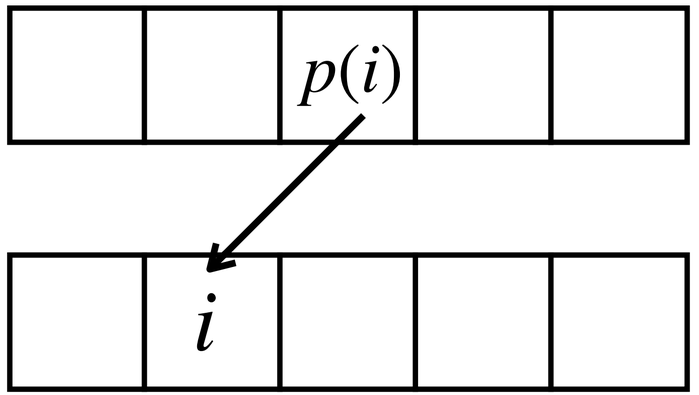
$$\left(\forall i\,.\,0 \le i < len(arr) - 1 \rightarrow out[i] \ge out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$$

$$\wedge\, \exists p : int \rightarrow int\,.\,\forall i\,.\left(0 \le i < len(out) \rightarrow \left(\underline{\exists j.0 \le j < len(arr)\,.\,p(j) = i} \wedge \underline{out[i] = arr[p(i)]}\right)\right)$$

But this alone is not enough, because multiple elements in output may be mapped from the same input



Every element in the output is mapped from some element in input



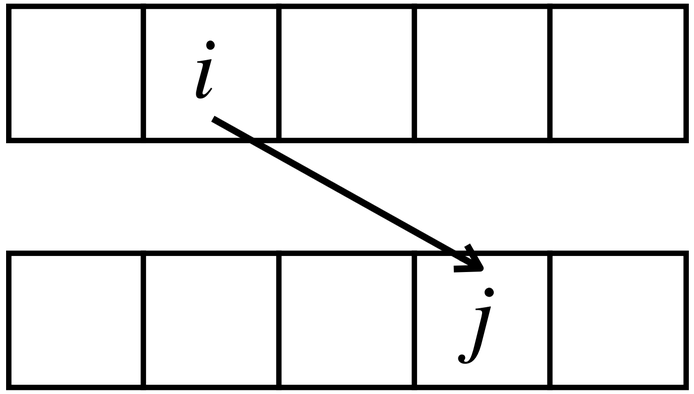
Pre and Postconditions

$$\left(\forall i\,.\,0 \le i < len(arr) - 1 \rightarrow out[i] \ge out[i+1]\right) \wedge \left(len(arr) = len(out)\right)$$

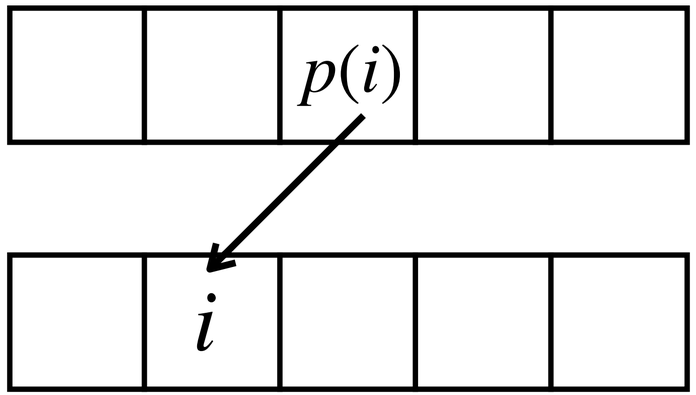
$$\wedge\, \exists p : int \rightarrow int\,.\,\forall i\,.\left(0 \le i < len(out) \rightarrow \left(\underline{\exists j.0 \le j < len(arr)\,.\,p(j) = i} \wedge \underline{out[i] = arr[p(i)]}\right)\right)$$

Every element in input maps to some element in output

$i = p(j)$



Every element in the output comes from some element in input



Pre and Postconditions Recap

- Take-away:
  - Pre/post can fully specify a program
  - But it’s not always easy to write pre and post

Program Synthesis From Pre/Postconditions

- Once we can specify program behaviors using pre and post
  - Verification: given a program and a spec, check if program satisfies spec
  - Synthesis: given a spec, generate program that satisfies spec

Program Synthesis From Pre/Postconditions

- Once we can specify program behaviors using pre and post
  - Verification: given a program and a spec, check if program satisfies spec
  - Synthesis: given a spec, generate program that satisfies spec

- Today's lecture: synthesis
  - Counterexample Guided Inductive Synthesis (CEGIS)
  - CEGIS relies on verification

Program Verification

- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post

Program Verification

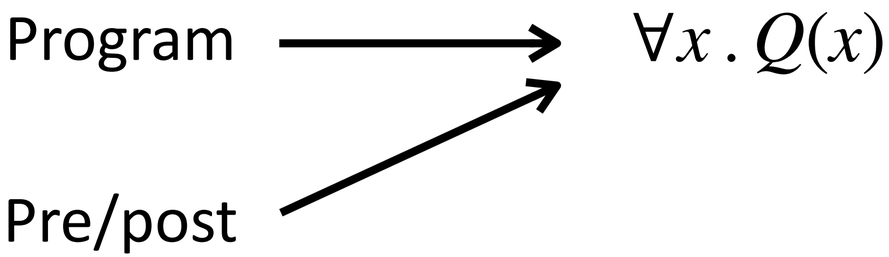
- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post
- Well studied problem, many techniques
  - Abstract Interpretation
  - Model Checking
  - Symbolic Execution
  - Etc.

Program Verification

- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post
- Well studied problem, many techniques
  - Abstract Interpretation
  - Model Checking
  - Symbolic Execution
  - Etc.
- One popular approach: constraint-based approach

Constraint-based Program Verification

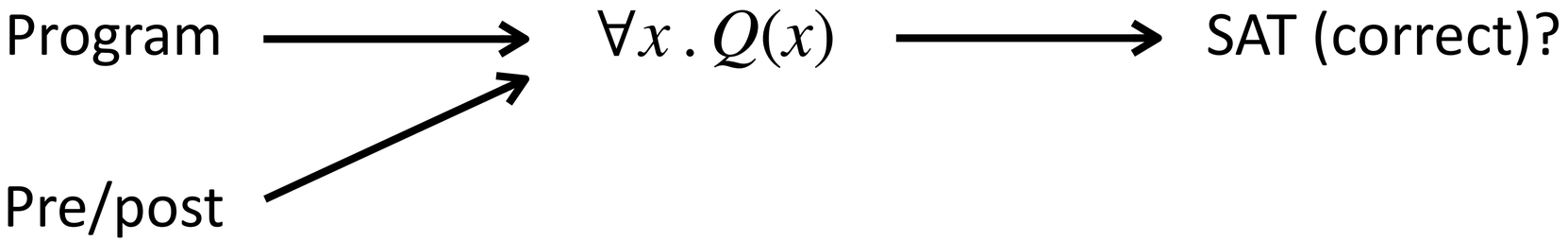
- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post
- Basic idea: convert P, pre, and post into a logical formula $\phi : \forall x \,.\, Q(x)$



Constraint-based Program Verification

- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post

- Basic idea: convert P, pre, and post into a logical formula $\phi : \forall x \,.\, Q(x)$
  - If ϕ is valid, that means P satisfies pre/post
  - Otherwise, P does not satisfy pre/post



Constraint-based Program Verification

- Goal: given program P, a precondition, and a postcondition, prove for that all inputs x satisfying pre, $P(x)$ satisfies post

- Basic idea: convert P, pre, and post into a logical formula $\phi : \forall x \,.\, Q(x)$
  - If ϕ is valid, that means P satisfies pre/post
  - Otherwise, P does not satisfy pre/post

- We have pretty good SMT solvers for solving formulas of the form $\forall x \,.\, Q(x)$
- Many techniques are based on this idea: model checking, symbolic execution, …
- We will not go into details

From Program Verification to Program Synthesis

- We can also view program synthesis as a constraint solving problem

From Program Verification to Program Synthesis

- We can also view program synthesis as a constraint solving problem
  - Solving formula: $\exists P \,.\, \forall x \,.\, Q(P, x)$
  - “Find a P, such that for all inputs x, P satisfies spec”

From Program Verification to Program Synthesis

- We can also view program synthesis as a constraint solving problem
  - Solving formula: $\exists P \,.\, \forall x \,.\, Q(P, x)$
  - “Find a P, such that for all inputs x, P satisfies spec”
  - Can we solve synthesis just like we solve program verification?

Solving Program Synthesis By Solving Quantifier Alternation

- Can we solve synthesis just like we solve program verification?

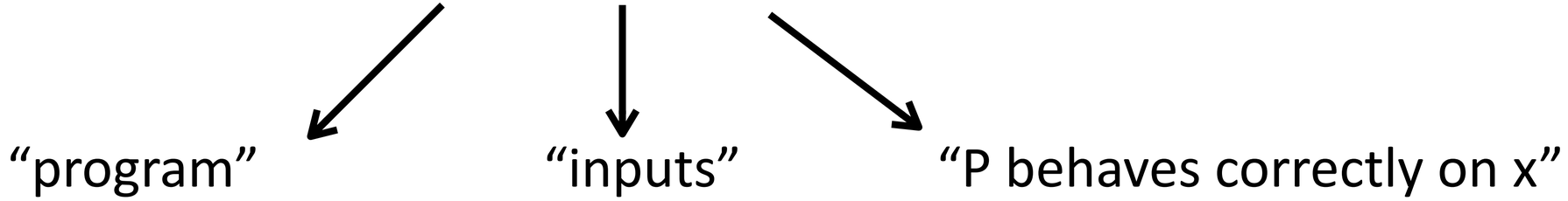
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$

“program” “inputs” “P behaves correctly on x”

Solving Program Synthesis By Solving Quantifier Alternation

- Can we solve synthesis just like we solve program verification?

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

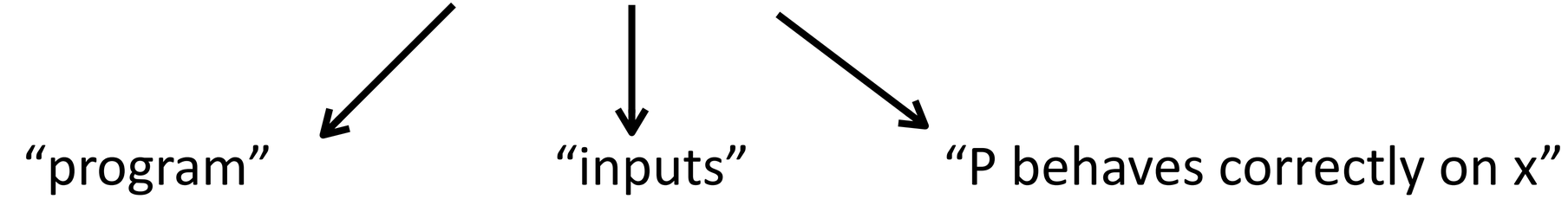


- Yes, if one can solve this formula

Solving Program Synthesis By Solving Quantifier Alternation

- Can we solve synthesis just like we solve program verification?

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



- Yes, if one can solve this formula
- But this is significantly more challenging compared to solving $\forall x \,.\, Q(P, x)$
  - Because it involves quantifier alternation
  - CEGIS comes to rescue!

Naive Solution

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

- Naive solution
  - Enumerate candidate programs, check each candidate, return first one that's correct

Naive Solution

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

- Naive solution
  - Enumerate candidate programs, check each candidate, return first one that's correct
  - Would this work (and why)? What do we need for it to work?

Naive Solution

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

- Naive solution
  - Enumerate candidate programs, check each candidate, return first one that's correct
  - Require a checker
  - … but each time invoking checker is slow
  - E.g., to enumerate 10K programs, need to invoke checker 10K times

Naive Solution

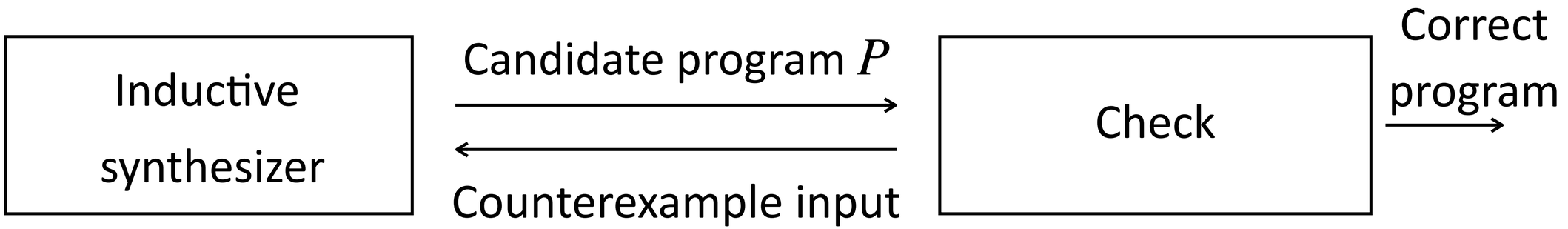
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$

- Naive solution
  - Enumerate candidate programs, check each candidate, return first one that's correct
  - Require a checker
  - … but each time invoking checker is slow
  - E.g., to enumerate 10K programs, need to invoke checker 10K times

- CEGIS idea: use a checker that can produce counterexamples!

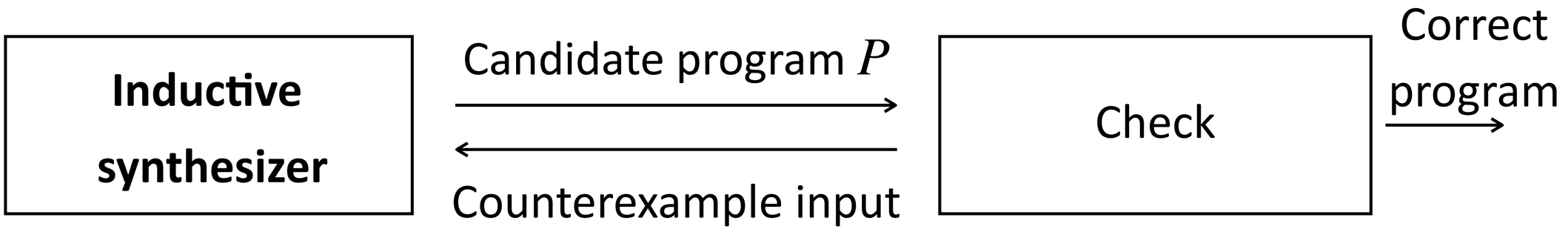
Counterexample Guided Inductive Synthesis (CEGIS)

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



Counterexample Guided Inductive Synthesis (CEGIS)

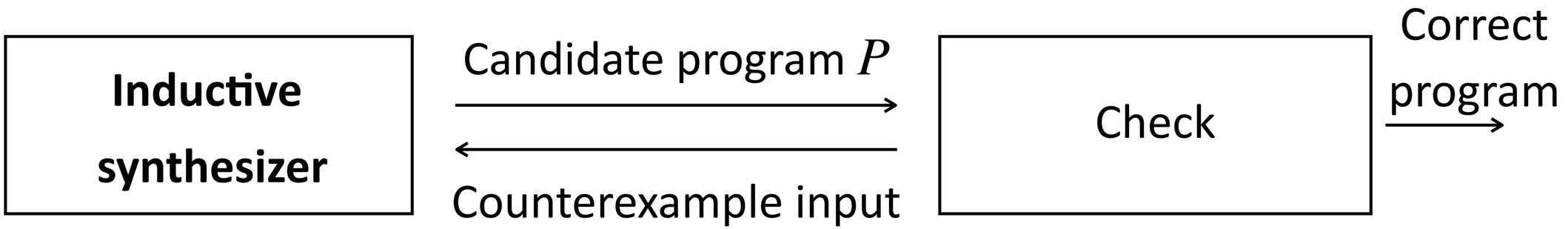
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Synthesize program inductively from a set of concrete inputs $x_1, \cdots, x_n$
  - Goal: find P s.t. $Q(P, x_1) \wedge \cdots \wedge Q(P, x_n)$

Counterexample Guided Inductive Synthesis (CEGIS)

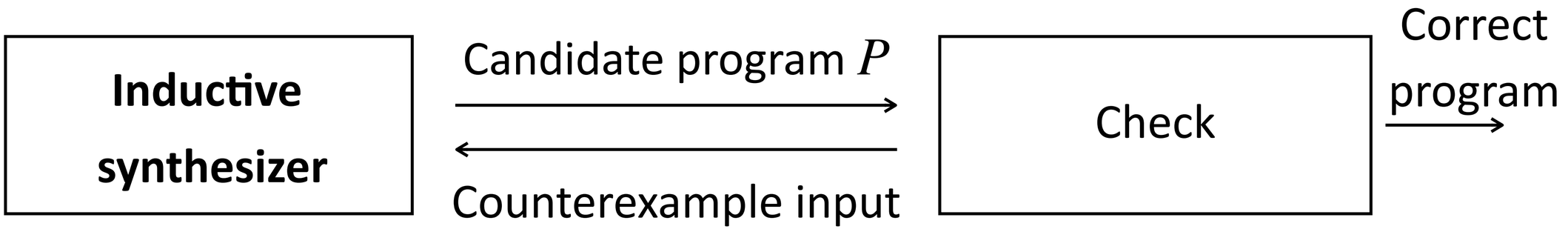
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Synthesize program inductively from a set of concrete inputs $x_1, \cdots, x_n$
  - Goal: find P s.t. $Q(P, x_1) \wedge \cdots \wedge Q(P, x_n)$
  - Basically, P behaves correctly on a finite number of inputs

Counterexample Guided Inductive Synthesis (CEGIS)

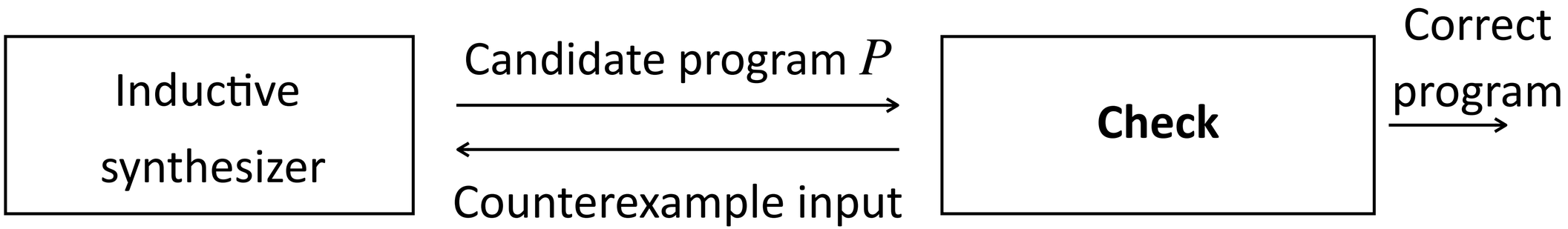
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Synthesize program inductively from a set of concrete inputs $x_1, \cdots, x_n$
  - Goal: find P s.t. $Q(P, x_1) \wedge \cdots \wedge Q(P, x_n)$
  - Basically, P behaves correctly on a finite number of inputs
  - Initially, no examples —> randomly guess one

Counterexample Guided Inductive Synthesis (CEGIS)

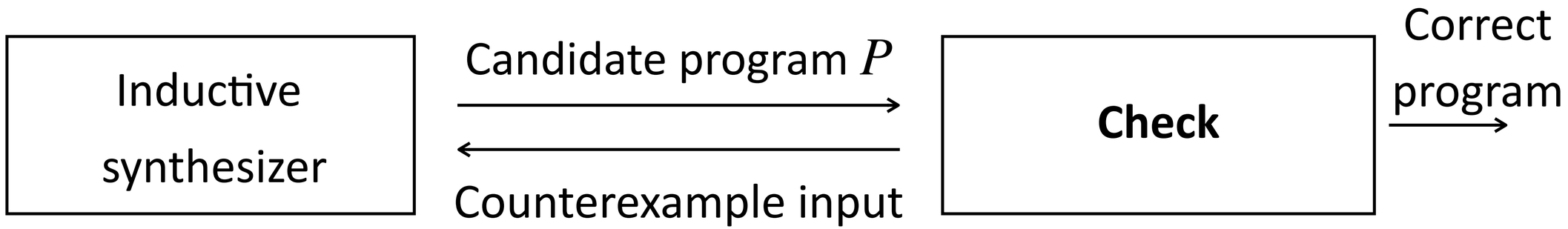
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Check whether P_n satisfies spec, if not, produce a **counterexample input**

Counterexample Guided Inductive Synthesis (CEGIS)

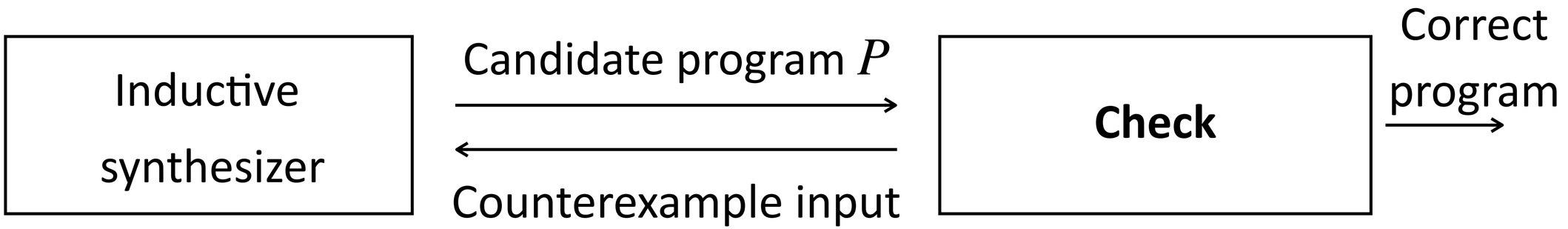
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Check whether P_n satisfies spec, if not, produce a **counterexample input**
  - Check validity: $\forall x \,.\, Q(P_n, x)$

Counterexample Guided Inductive Synthesis (CEGIS)

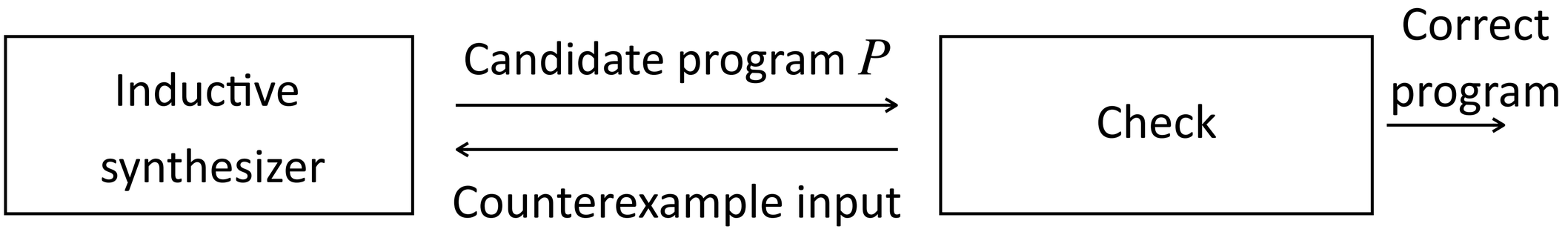
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Check whether P_n satisfies spec, if not, produce a **counterexample input**
  - Check validity: $\forall x \,.\, Q(P_n, x)$
  - If valid, P_n is solution; otherwise, P_n is wrong, produce a counterexample input x_{n+1}
    - Counterexample input: P fails but desired program should pass
    - $Q(P_n, x_{n+1}) = \bot$

Counterexample Guided Inductive Synthesis (CEGIS)

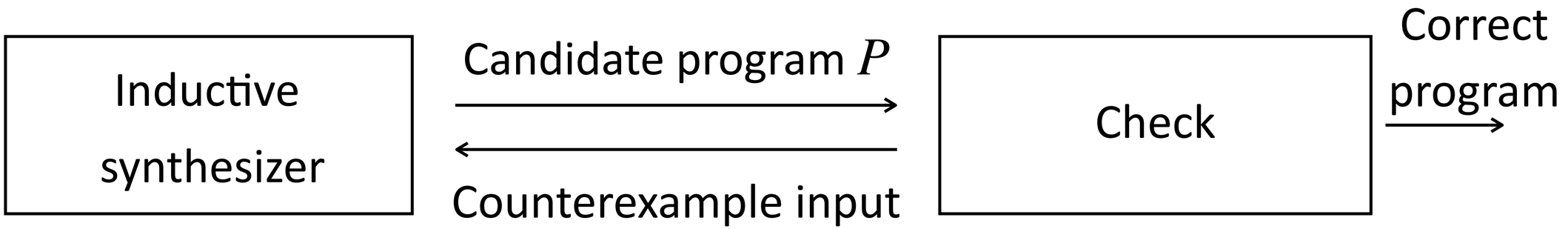
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Repeat inductive synthesis from a set of concrete inputs $x_1, \cdots, x_n, x_{n+1}$

Counterexample Guided Inductive Synthesis (CEGIS)

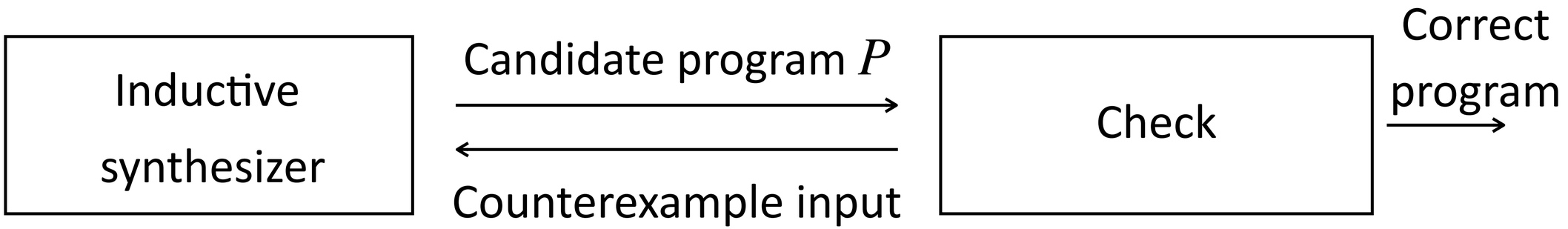
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Repeat inductive synthesis from a set of concrete inputs $x_1, \cdots, x_n, x_{n+1}$
- … util check says yes!

Counterexample Guided Inductive Synthesis (CEGIS)

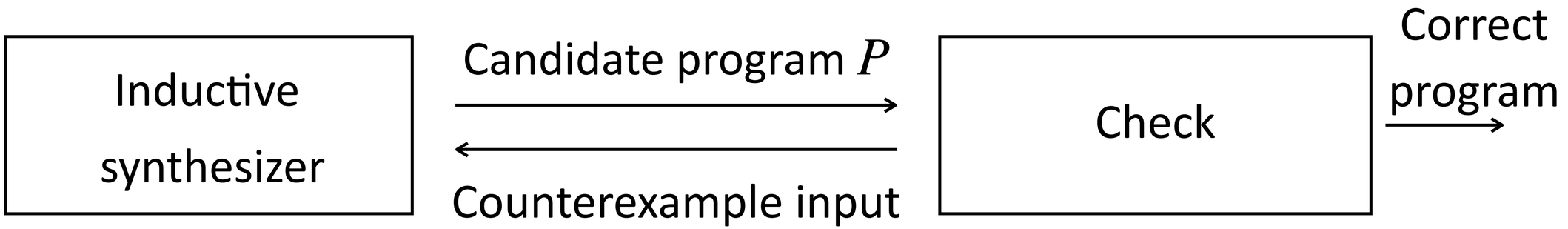
$$\exists P \,.\, \forall x \,.\, Q(P, x)$$



- Repeat inductive synthesis from a set of concrete inputs $x_1, \cdots, x_n, x_{n+1}$
- … util check says yes!
- … or inductive synthesizer exhausts search space

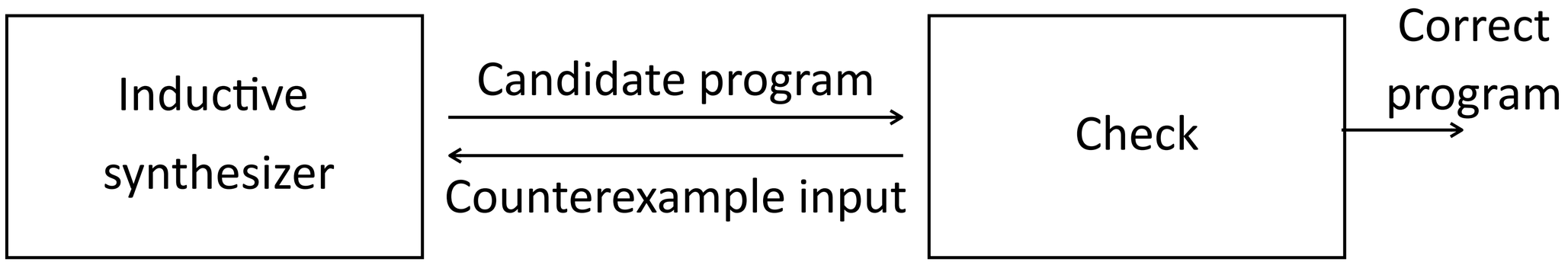
Counterexample Guided Inductive Synthesis (CEGIS)

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



- Repeat inductive synthesis from a set of concrete inputs $x_1, \cdots, x_n, x_{n+1}$
- … util check says yes!
- … or inductive synthesizer exhausts search space
- … or you run out of patience

Counterexample Guided Inductive Synthesis (CEGIS)



- $P(x, y)$ such that $\forall x\,.\,y\,.\;out \geq x \wedge out \geq y \wedge \left(out = x \vee out = y\right)$
- Target: $max(x, y)$

	Counterexample inputs	Candidate program
• Iteration 1	$\{\ \}$	x
• Iteration 2	$\{(1,2)\}$	y
• Iteration 3	$\{(1,2),(2,1)\}$	$max(x, y)$

CEGIS Recap

- Decompose $\exists P \,.\, \forall x \,.\, Q(P, x)$ into two steps

CEGIS Recap

- Decompose $\exists P \,.\, \forall x \,.\, Q(P, x)$ into two steps
  - Inductive synthesis from example inputs $x_1, \cdots, x_n$
  - Check P against $\forall x \,.\, Q(P, x)$, if failed, generate counterexample input

CEGIS Recap

- Decompose $\exists P \,.\, \forall x \,.\, Q(P, x)$ into two steps
  - Inductive synthesis from example inputs $x_1, \cdots, x_n$
    - Guarantee $Q(P, x_1) \wedge \cdots \wedge Q(P, x_n)$
  - Check P against $\forall x \,.\, Q(P, x)$, if failed, generate counterexample input

CEGIS Recap

- Decompose $\exists P \,.\, \forall x \,.\, Q(P, x)$ into two steps
  - Inductive synthesis from example inputs $x_1, \cdots, x_n$
    - Guarantee $Q(P, x_1) \wedge \cdots \wedge Q(P, x_n)$
  - Check P against $\forall x \,.\, Q(P, x)$, if failed, generate counterexample input

- Use counterexamples to guide inductive synthesis
  - Avoid producing programs that fail in similar ways from previous rejected programs

CEGIS Recap

- Allow to mix and match different inductive synthesizers with different checkers
  - Inductive synthesizers: search-based, representation-based, …
  - Checkers: model checking, testing, …

CEGIS Recap

- Allow to mix and match different inductive synthesizers with different checkers
  - Inductive synthesizers: search-based, representation-based, …
  - Checkers: model checking, testing, …
- Convergence not guaranteed

CEGIS Recap

- Allow to mix and match different inductive synthesizers with different checkers
  - Inductive synthesizers: search-based, representation-based, …
  - Checkers: model checking, testing, …

- Convergence not guaranteed
- In many problems, converge to correct program with a few iterations

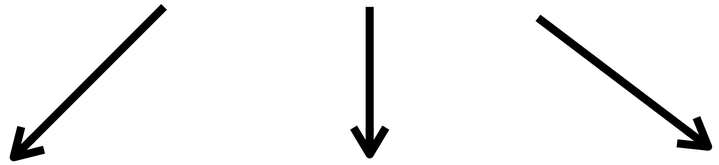
CEGIS Recap

- Allow to mix and match different inductive synthesizers with different checkers
  - Inductive synthesizers: search-based, representation-based, …
  - Checkers: model checking, testing, …

- Convergence not guaranteed
- In many problems, converge to correct program with a few iterations
- However, there are known cases where CEGIS fails badly
- If search space is unbounded, may not terminate

CEGIS Recap

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

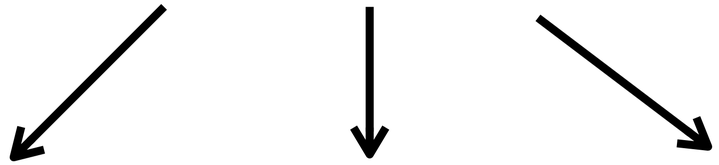


“program” “inputs” “P behaves correctly on x”

- Think about CEGIS broadly

CEGIS Recap

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

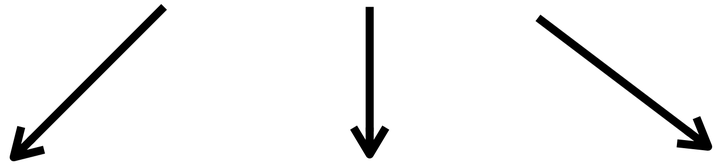


"program" "inputs" "P behaves correctly on x"

- Think about CEGIS broadly
  - Specification doesn't have to be formulas
    - Reference implementation: $\exists P \,.\, Q(P', P)$ (check equivalence between P and P')

CEGIS Recap

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

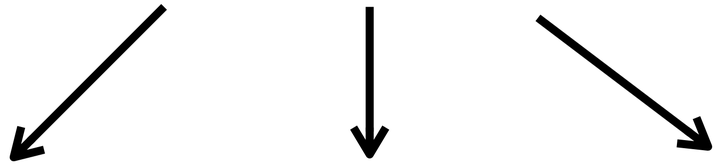


"program" "inputs" "P behaves correctly on x"

- Think about CEGIS broadly
  - Specification doesn't have to be formulas
    - Reference implementation: $\exists P \,.\, Q(P', P)$ (check equivalence between P and P')
  - Checker can be testing-based

CEGIS Recap

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



"program" "inputs" "P behaves correctly on x"

- Think about CEGIS broadly
  - Specification doesn't have to be formulas
    - Reference implementation: $\exists P \,.\, Q(P', P)$ (check equivalence between P and P')
  - Checker can be testing-based
  - Programs can be SQL queries, etc.