Computer-Aided Reasoning for Software

CSE507

Program Synthesis

Emina Torlak
emina@cs.washington.edu

Today

Last lecture

- Solvers as angelic runtime oracle

Today

- Program synthesis: from specs to code

Reminders

- HW3 is due on Friday.

topics

The program synthesis problem

$$\exists P. \forall x. \; \phi(x, P(x))$$

Find a program P that satisfies the specification ϕ on all inputs.

The program synthesis problem

ϕ may be a formula, a reference implementation, input / output pairs, traces, demonstrations, etc.

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

Find a program P that satisfies the specification ϕ on all inputs.

The program synthesis problem

ϕ may be a formula, a reference implementation, input / output pairs, traces, demonstrations, etc.

Synthesis improves

- Productivity (when writing ϕ is easier than writing P).
- Correctness (when verifying ϕ is easier than verifying P).

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

Find a program P that satisfies the specification ϕ on all inputs.

Two kinds of program synthesis

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

Synthesis as a problem in deductive theorem proving.

Synthesis as a search problem.

Two kinds of program synthesis

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

Deductive (classic) synthesis

Inductive (syntax-guided) synthesis

Two kinds of program synthesis

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

Deductive (classic) synthesis

Derive the program P from the constructive proof of the theorem $\forall$ x. $\exists$ y. $\phi(x, y)$.

Inductive (syntax-guided) synthesis

Two kinds of program synthesis

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

Deductive (classic) synthesis

Derive the program P from the constructive proof of the theorem $\forall x. \exists y. \phi(x, y)$.

Inductive (syntax-guided) synthesis

Discover the program P by searching a restricted space of candidate programs for one that satisfies ϕ on all inputs.

Two kinds of program synthesis

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

SPIRAL

Deductive (classic) synthesis

Derive the program P from the constructive proof of the theorem $\forall$ x. $\exists$ y. $\phi(x, y)$.

FlashFill

Inductive (syntax-guided) synthesis

Discover the program P by searching a restricted space of candidate programs for one that satisfies ϕ on all inputs.

Deductive synthesis with axioms and E-graphs

Denali Superoptimizer
[Joshi, Nelson, Randall, PLDI'02]

Deductive synthesis with axioms and E-graphs

Specification ϕ, given as a reference implementation.

```
reg6 * 4 + 1
```

Denali Superoptimizer [Joshi, Nelson, Randall, PLDI'02]

Deductive synthesis with axioms and E-graphs

Specification ϕ, given as a reference implementation.

Optimal (lowest cost) program P that is equivalent to ϕ on all inputs (values of reg6).

`reg6 * 4 + 1` → Denali Superoptimizer [Joshi, Nelson, Randall, PLDI'02] → `s4addl(reg6, 1)`

Deductive synthesis with axioms and E-graphs

Specification ϕ, given as a reference implementation.

```
reg6 * 4 + 1
```

Denali Superoptimizer [Joshi, Nelson, Randall, PLDI'02]

Optimal (lowest cost) program P that is equivalent to ϕ on all inputs (values of reg6).

```
s4addl(reg6, 1)
```

$\forall$ k, n. 2^n = 2**n

$\forall$ k, n. k*2^n = k << n

$\forall$ k, n. k*4 + n = **`s4addl`**(k, n)

…

Two kinds of axioms:

- Instruction semantics.
- Algebraic properties of functions and relations used for specifying instruction semantics.

Deductive synthesis with axioms and E-graphs

Specification ϕ, given as a reference implementation.

`reg6 * 4 + 1`

Denali Superoptimizer [Joshi, Nelson, Randall, PLDI'02]

1. Construct an E-graph.
2. Use a SAT solver to search the E-graph for a K-cycle program.

`s4addl(reg6, 1)`

Optimal (lowest cost) program P that is equivalent to ϕ on all inputs (values of reg6).

$\forall$ k, n. 2^n = 2**n

$\forall$ k, n. k*2^n = k << n

$\forall$ k, n. k*4 + n = **`s4addl`**(k, n)

...

Two kinds of axioms:

- Instruction semantics.
- Algebraic properties of functions and relations used for specifying instruction semantics.

Denali by example

`reg6 * 4 + 1`

$\forall$ k, n. 2^n = 2**n

$\forall$ k, n. k*2^n = k **<<** n

$\forall$ k, n. k*4 + n = **`s4addl`**(k, n)

…

E-graph matching

SAT

`s``4addl(reg6`**`,`**` 1)`

Denali by example

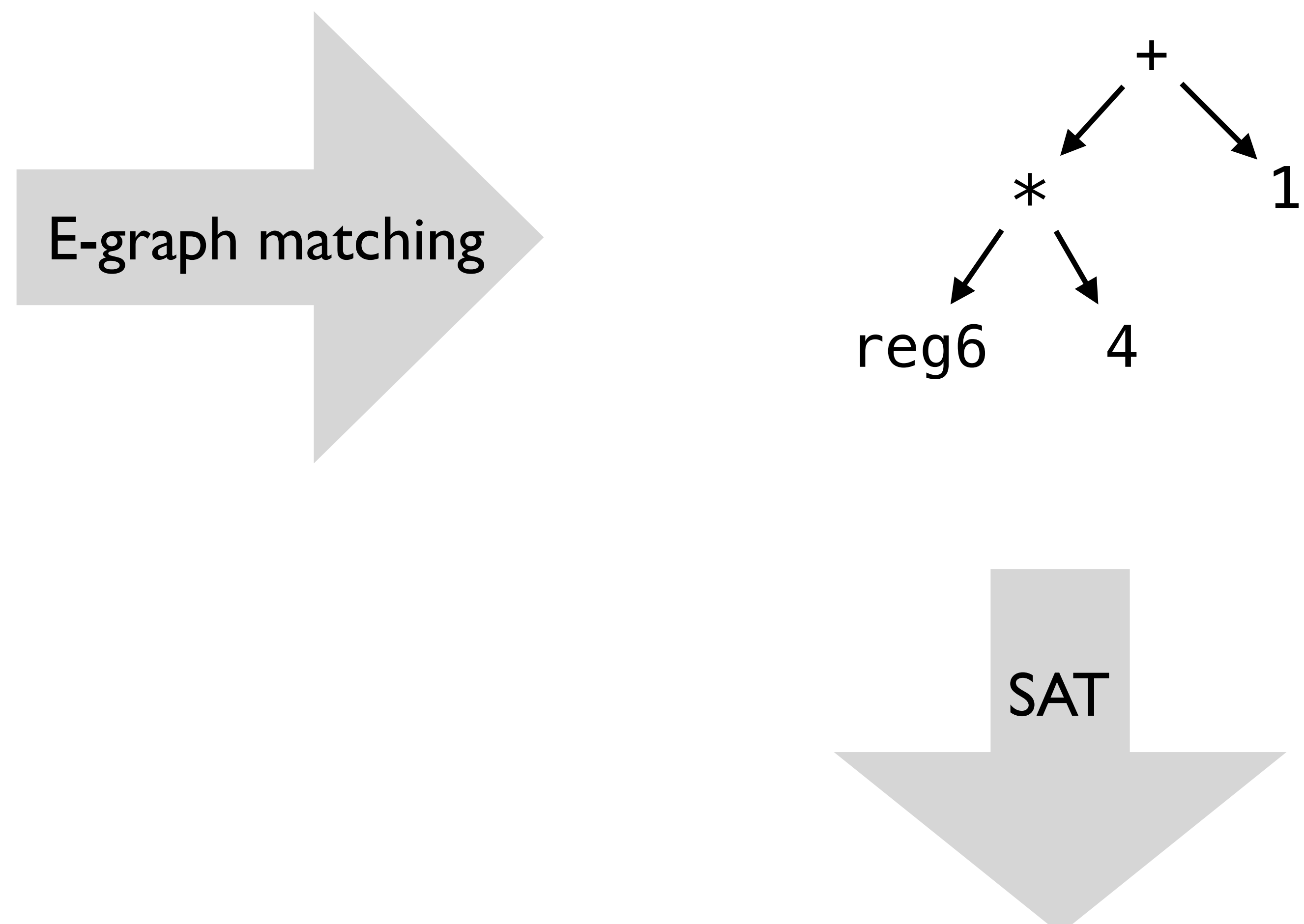
`reg6 * 4 + 1`

$\forall$ k, n. 2^n = 2**n

$\forall$ k, n. k*2^n = k **<<** n

$\forall$ k, n. k*4 + n = **`s4addl`**(k, n)

…



`s``4addl(reg6, 1)`

Denali by example

`reg6 * 4 + 1`

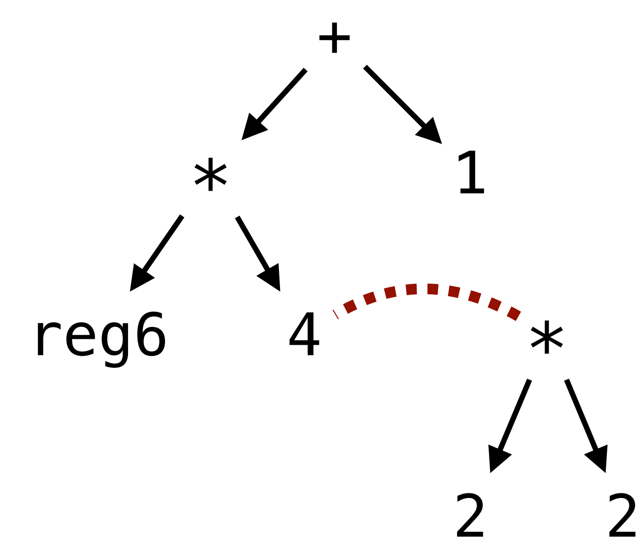
∀ k, n. 2^n = 2**n

∀ k, n. k*2^n = k **<<** n

∀ k, n. k*4 + n = **`s4addl`**(k, n)

…

E-graph matching



SAT

s4addl(reg6**,** 1)

Denali by example

`reg6 * 4 + 1`

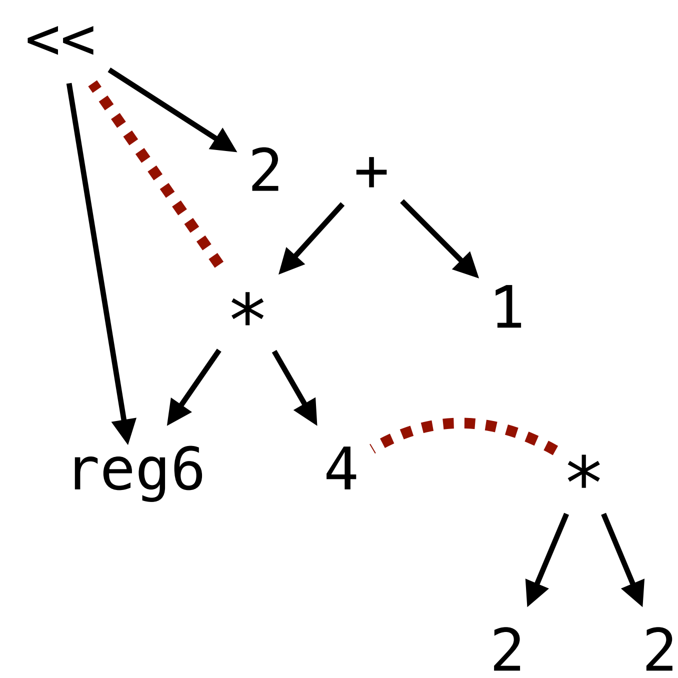
∀ k, n. 2^n = 2**n

∀ k, n. k*2^n = k **<<** n

∀ k, n. k*4 + n = **`s4addl`**(k, n)

…

E-graph matching



SAT

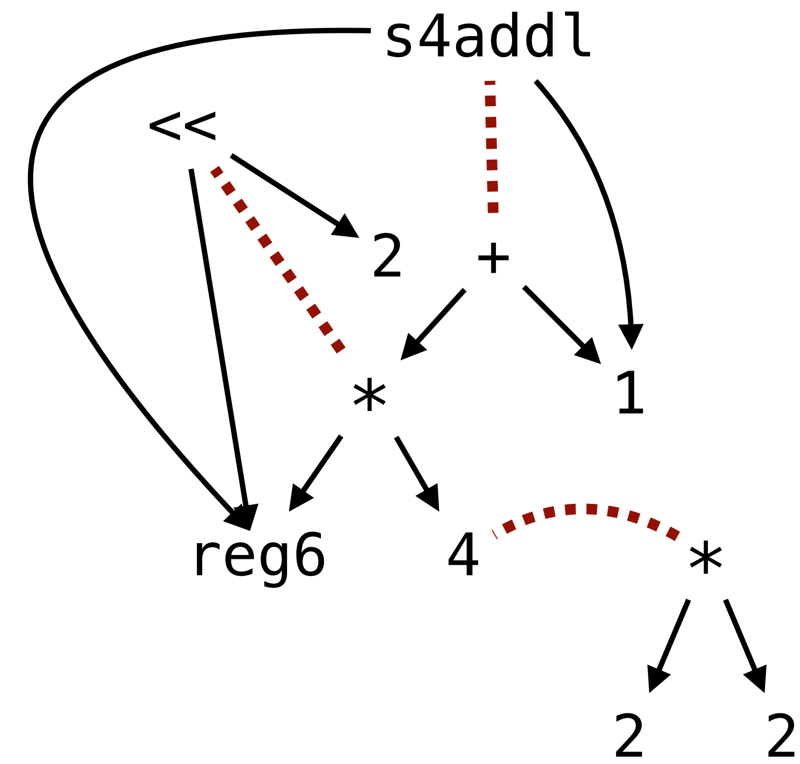
s4addl(reg6, 1)

Denali by example

`reg6 * 4 + 1`

- $\forall$ k, n. 2^n = 2**n
- $\forall$ k, n. k*2^n = k **<<** n
- $\forall$ k, n. k*4 + n = **`s4addl`**(k, n)
- …

E-graph matching



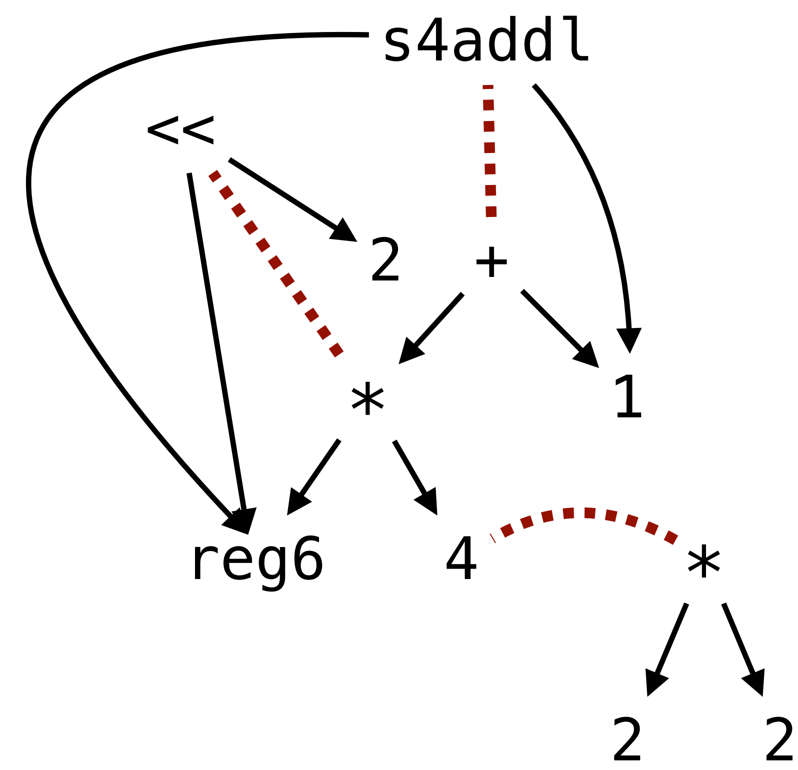
SAT

`s``4addl(reg6, 1)`

Deductive synthesis versus compilation

Deductive synthesizer

- Non-deterministic.
- Searches *all correct rewrites* for one that is optimal.



Compiler

- Deterministic.
- Lowers a source program into a target program using a *fixed sequence of rewrite steps*.

```
reg6 * 4 + 1
     ↓
reg6 << 2 + 1
```

Deductive synthesis versus inductive synthesis

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

Deductive synthesis

- Efficient and provably correct: thanks to the semantics-preserving rules, only correct programs are explored.
- Requires *sufficient axiomatization* of the domain.
- Requires *complete* specifications to seed the derivation.

Deductive synthesis versus inductive synthesis

$\exists$ P. $\forall$ x. ϕ(x, P(x))

Deductive synthesis

- Efficient and provably correct: thanks to the semantics-preserving rules, only correct programs are explored.
- Requires *sufficient axiomatization* of the domain.
- Requires *complete* specifications to seed the derivation.

Inductive synthesis

- Works with *multi-modal and partial* specifications.
- Requires *no axioms*.
- But often at the cost of *lower efficiency* and *weaker (bounded) guarantees* on the correctness/ optimality of synthesized code.

Inductive syntax-guided synthesis

CEGIS:
Counterexample-Guided
Inductive Synthesis
[**Solar-Lezama et al, ASPLOS'06**]

Inductive syntax-guided synthesis

A partial or multimodal specification ϕ of the desired program (e.g., assertions, i/o pairs).

```
reg6 * 4 + 1
```

CEGIS: Counterexample-Guided Inductive Synthesis [Solar-Lezama et al, ASPLOS'06]

Inductive syntax-guided synthesis

A partial or multimodal specification ϕ of the desired program (e.g., assertions, i/o pairs).

```
reg6 * 4 + 1
```

CEGIS: Counterexample-Guided Inductive Synthesis [**Solar-Lezama et al, ASPLOS'06**]

```
expr :=
 const | reg6 |
 s4addl(expr, expr) |
 …
```

A syntactic *sketch* (e.g., a grammar) describing the shape of the desired program P.

This defines the space of candidate programs to search. Can be fine-tuned for better performance.

Inductive syntax-guided synthesis

A partial or multimodal specification ϕ of the desired program (e.g., assertions, i/o pairs).

```
reg6 * 4 + 1
```

CEGIS: Counterexample-Guided Inductive Synthesis [Solar-Lezama et al, ASPLOS'06]

A program P from the given space of candidates that satisfies ϕ on all (usually bounded) inputs.

```
s4addl(reg6, 1)
```

```
expr :=
 const | reg6 |
 s4addl(expr, expr) |
 …
```

A syntactic *sketch* (e.g., a grammar) describing the shape of the desired program P.

This defines the space of candidate programs to search. Can be fine-tuned for better performance.

Inductive syntax-guided synthesis

A partial or multimodal specification ϕ of the desired program (e.g., assertions, i/o pairs).

Guess a program that works on a finite set of inputs, verify it, and learn from bad guesses.

A program P from the given space of candidates that satisfies ϕ on all (usually bounded) inputs.

`reg6 * 4 + 1`

CEGIS:
Counterexample-Guided Inductive Synthesis
[Solar-Lezama et al, ASPLOS'06]

`s4addl(reg6, 1)`

```
expr :=
  const | reg6 |
  s4addl(expr, expr) |
  …
```

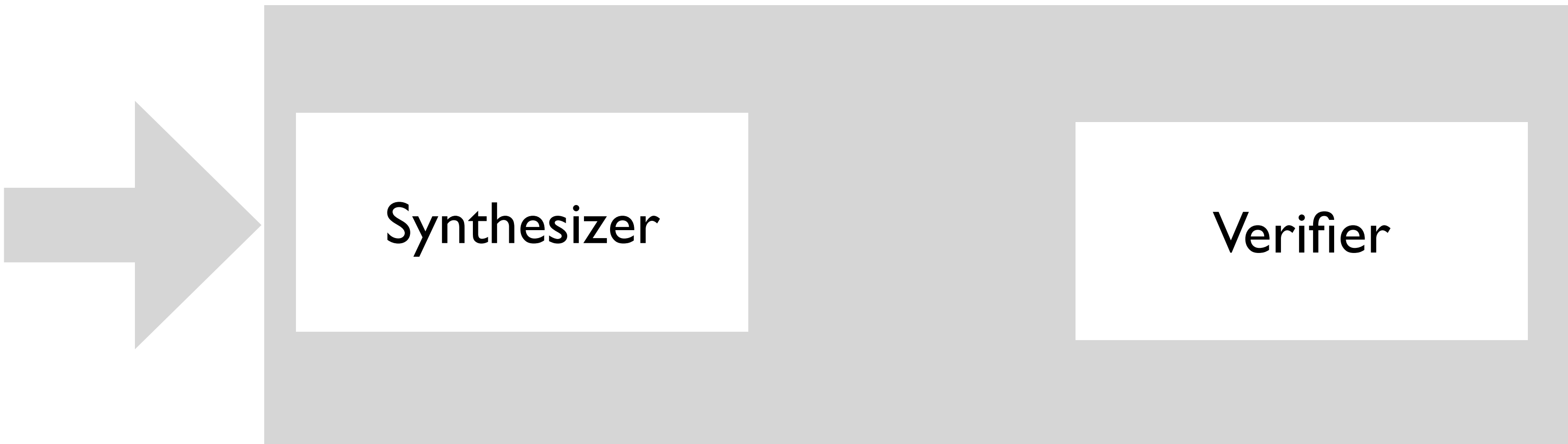
A syntactic *sketch* (e.g., a grammar) describing the shape of the desired program P.

This defines the space of candidate programs to search. Can be fine-tuned for better performance.

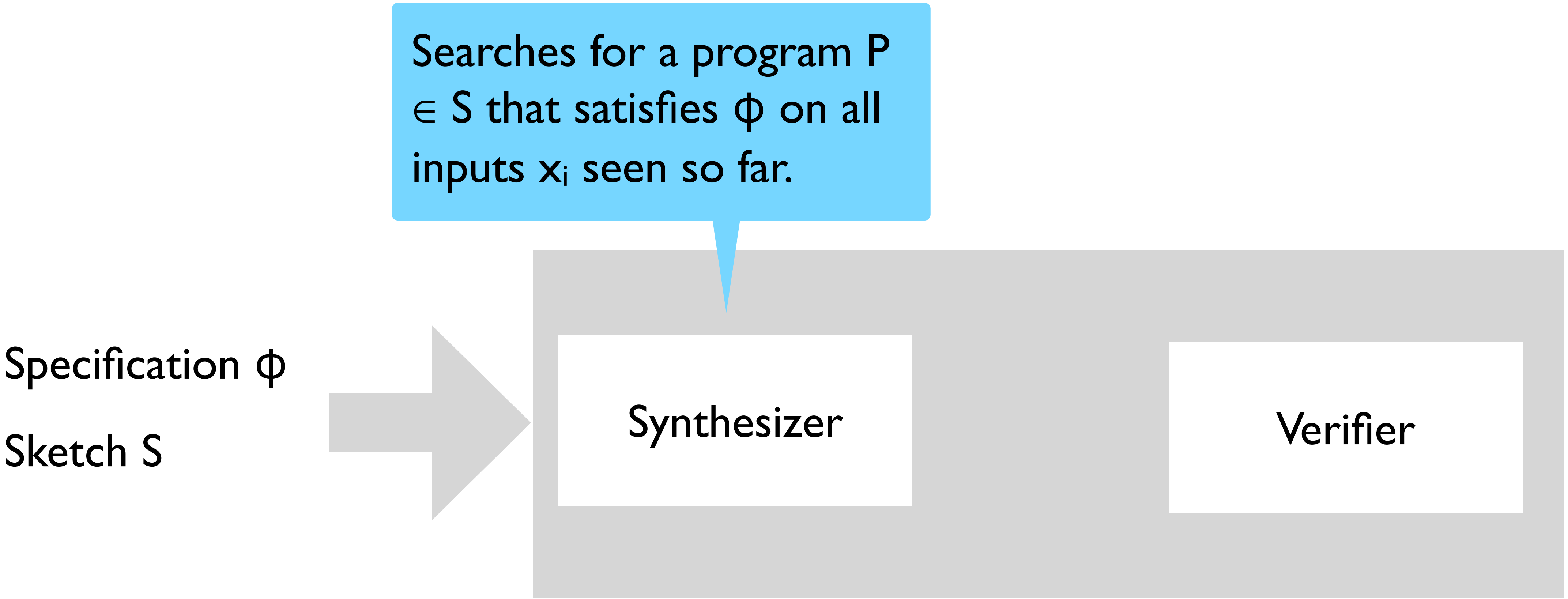
Overview of CEGIS

Specification ϕ

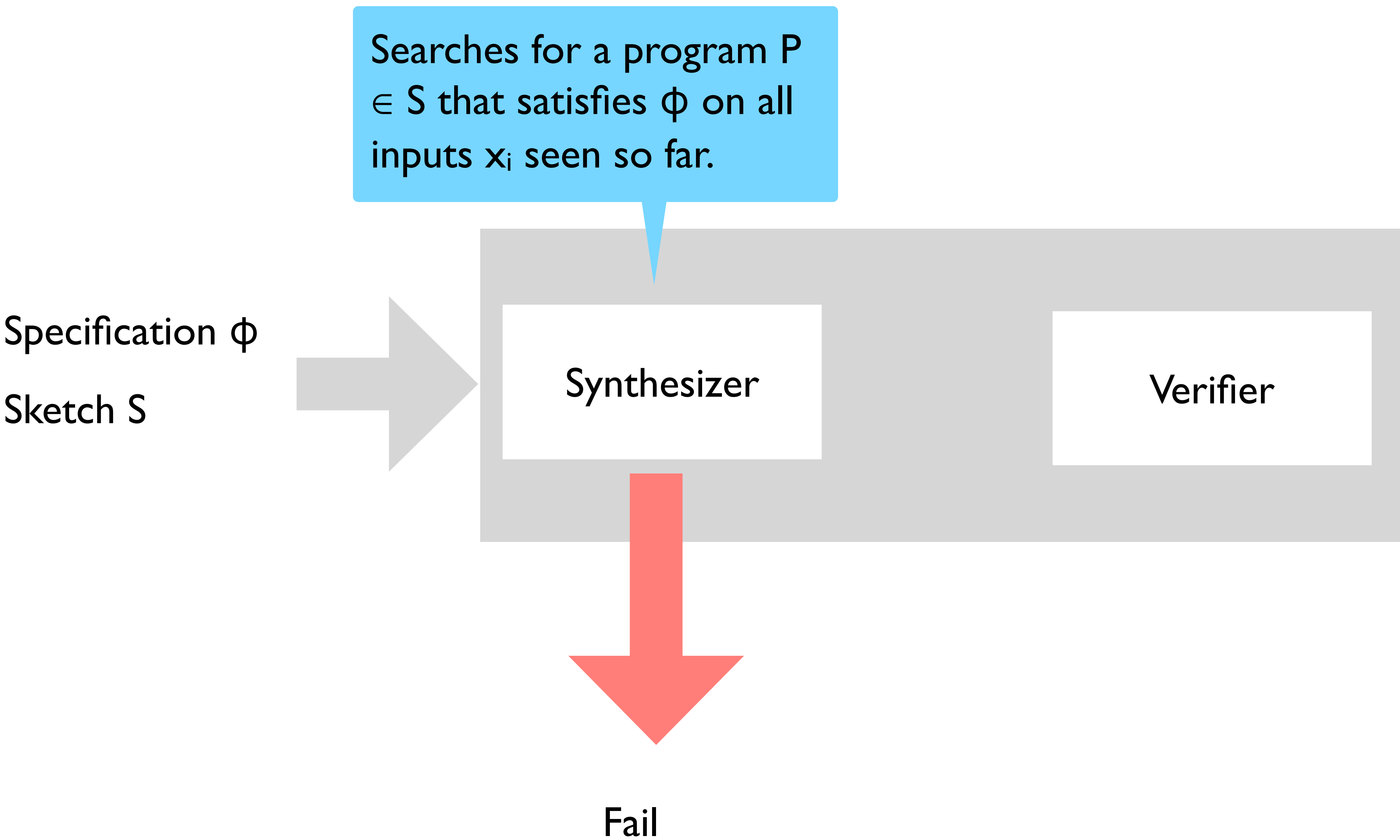
Sketch S



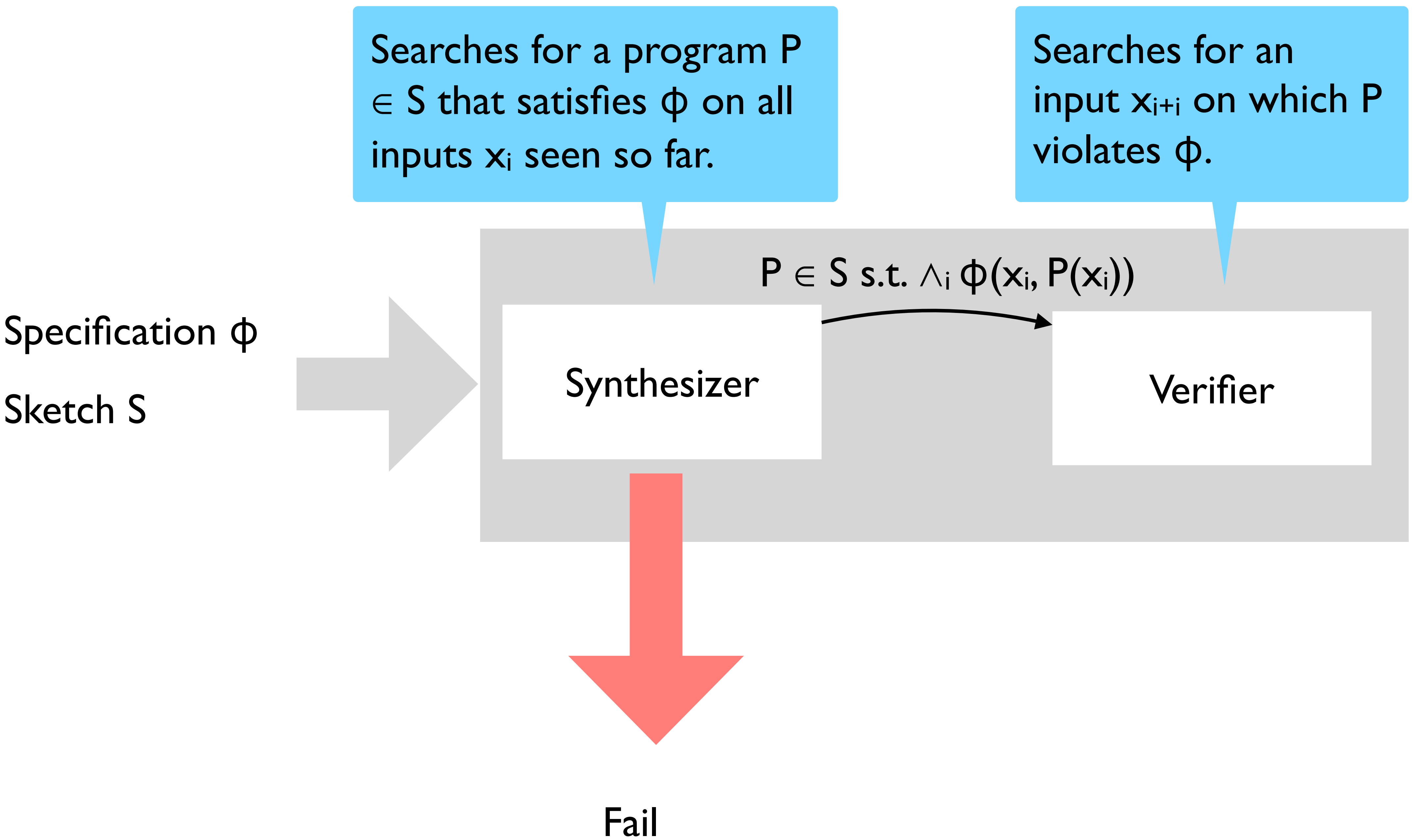
Overview of CEGIS



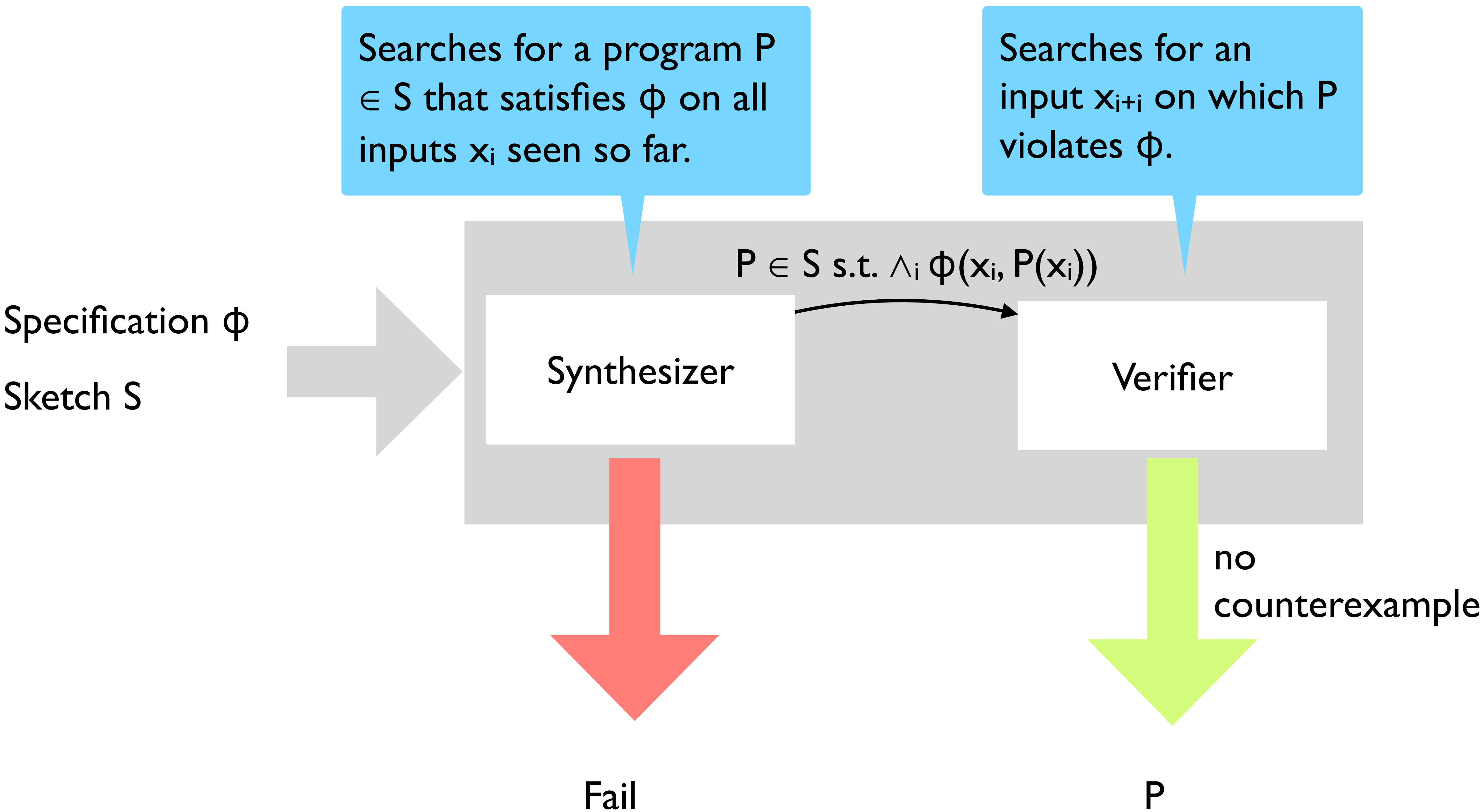
Overview of CEGIS



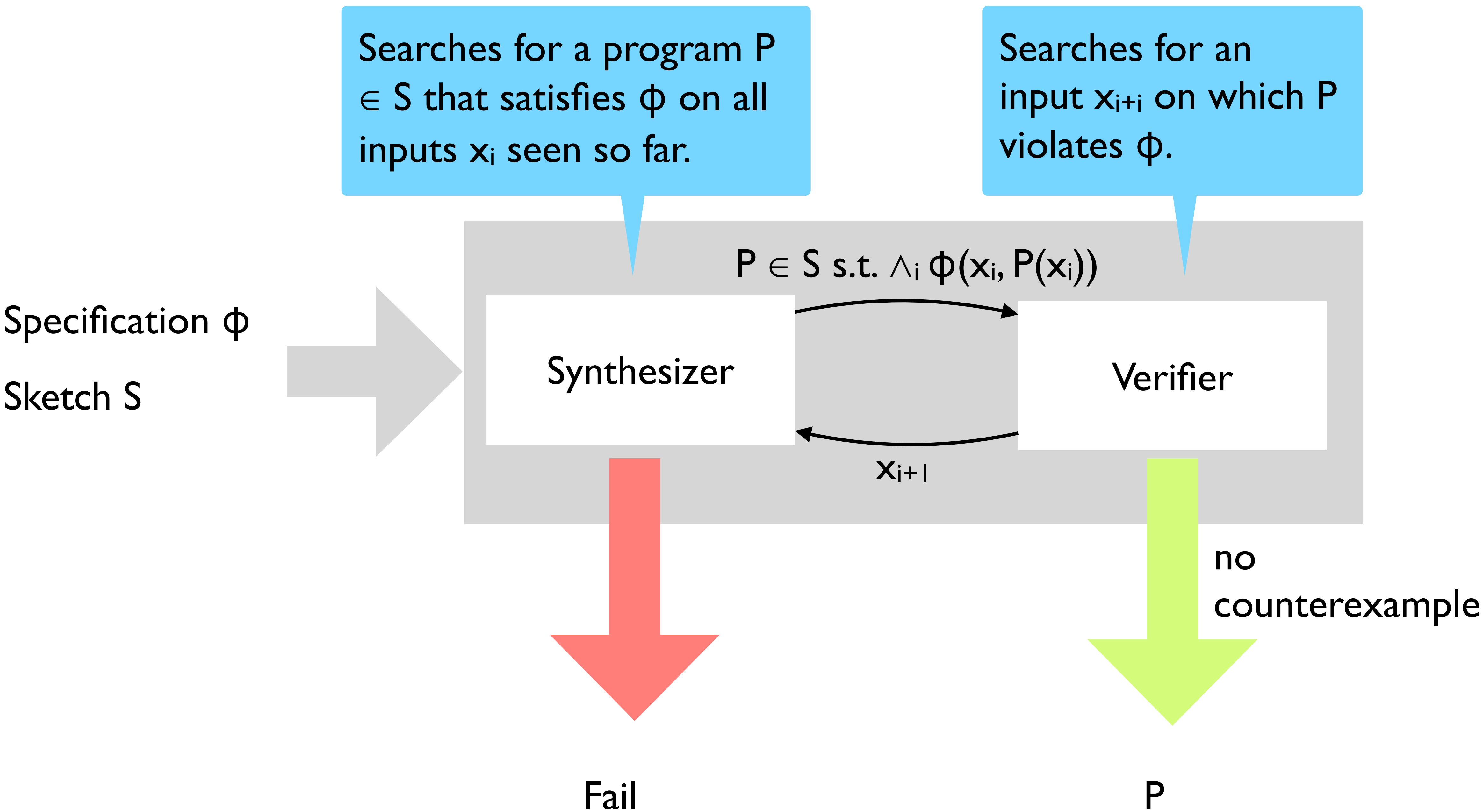
Overview of CEGIS



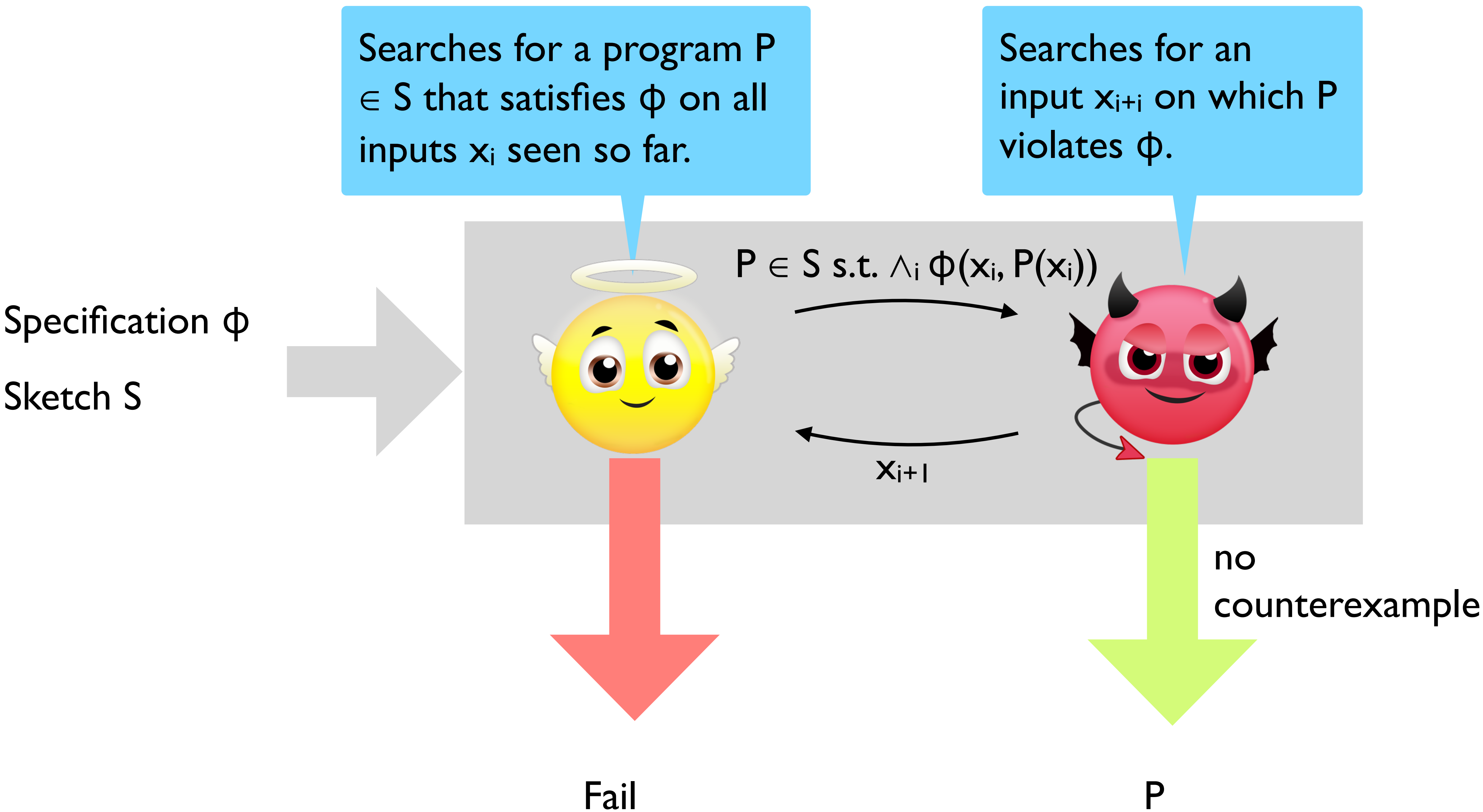
Overview of CEGIS



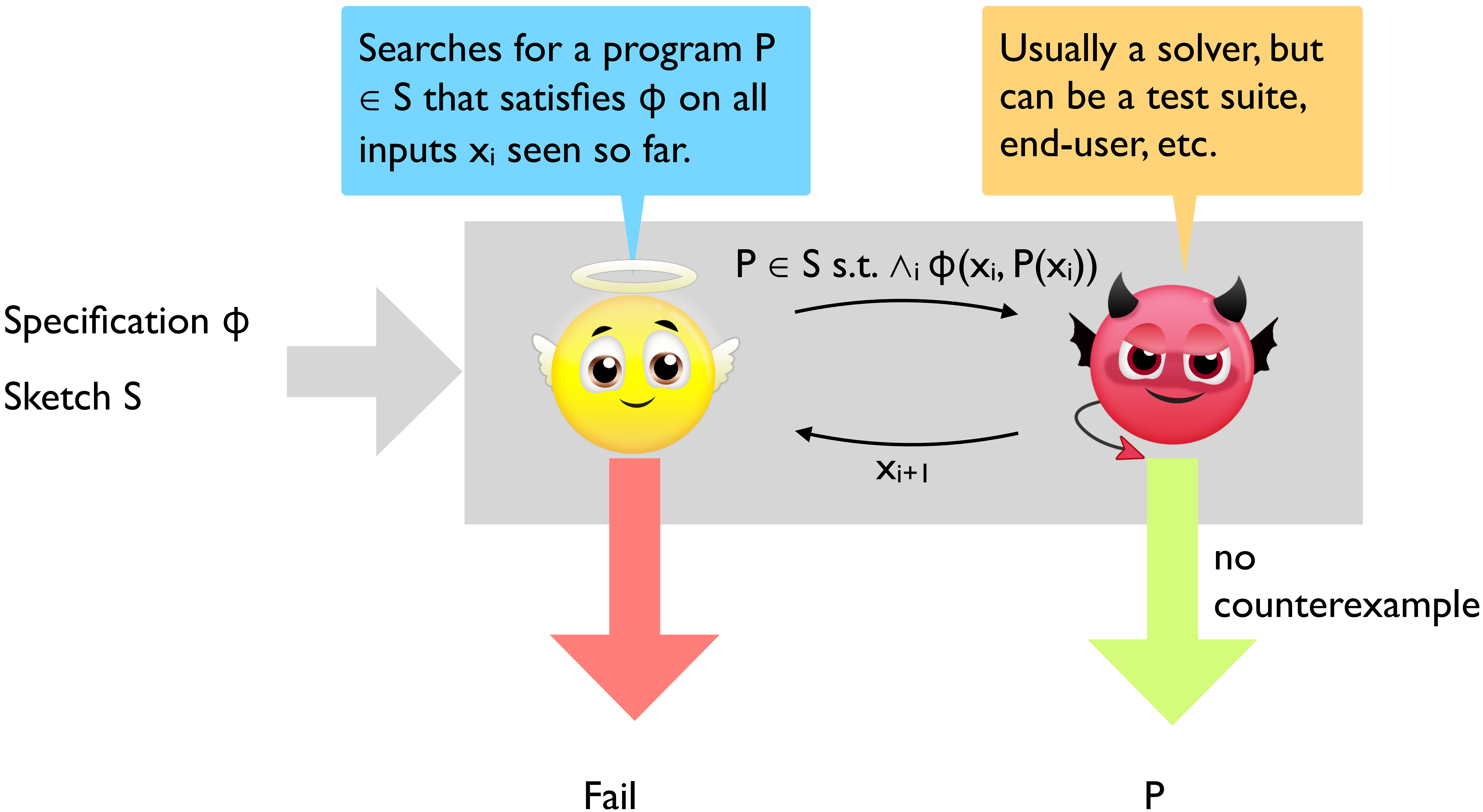
Overview of CEGIS



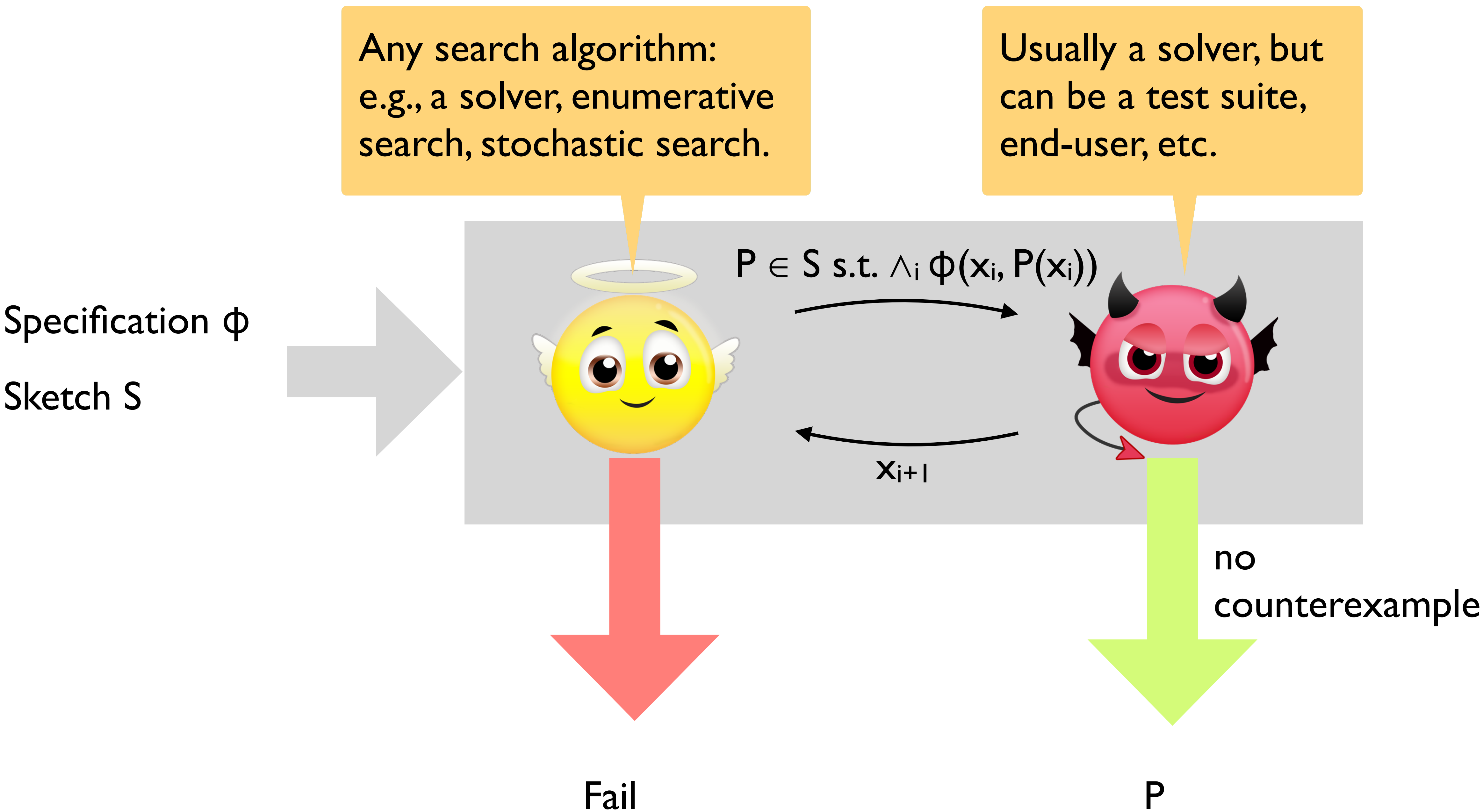
Overview of CEGIS



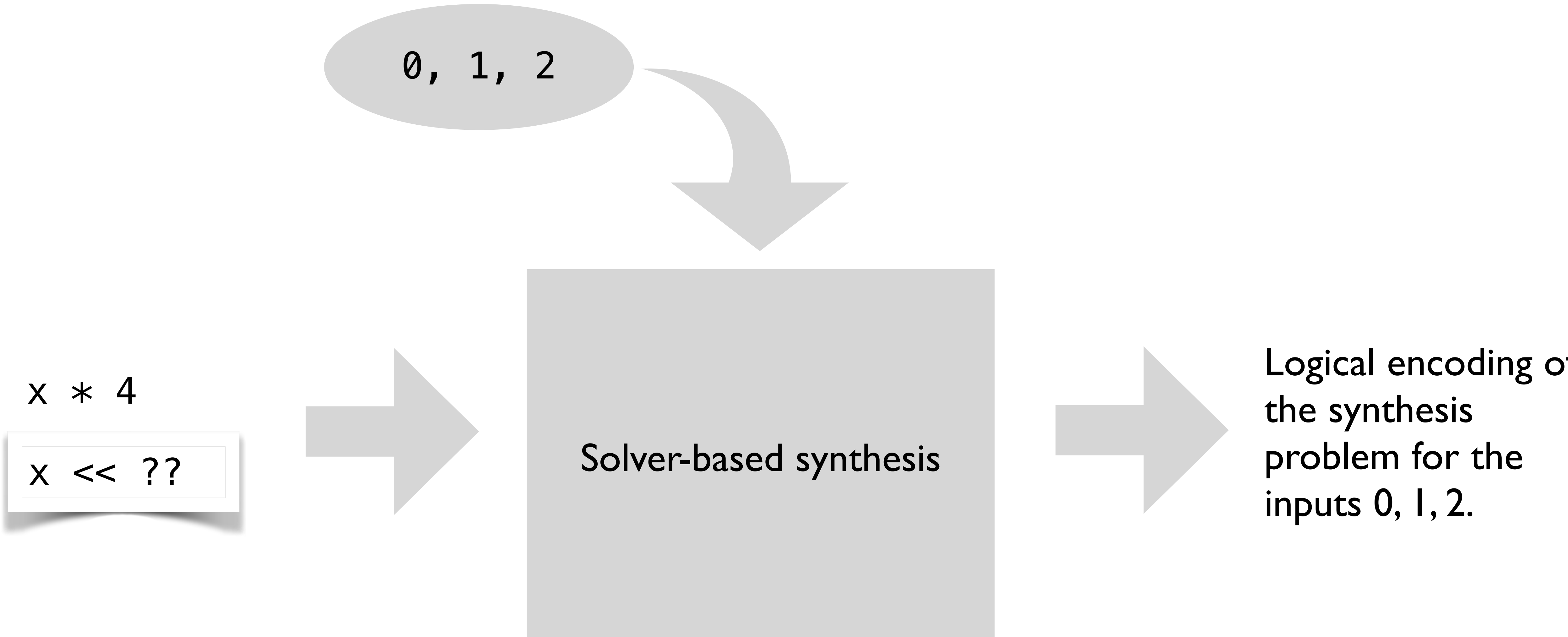
Overview of CEGIS



Overview of CEGIS



Synthesizing programs with a solver



[Solar-Lezama et al, ASPLOS'06]

Synthesizing programs with a solver

`0, 1, 2`

`x * 4`

`x << n`

- Replace each ?? with a fresh symbolic constant.

Logical encoding of the synthesis problem for the inputs 0, 1, 2.

[Solar-Lezama et al, ASPLOS'06]

Synthesizing programs with a solver

```
0, 1, 2
```

```
x * 4
```

```
x << n
```

- Replace each ?? with a fresh symbolic constant.
- Translate the resulting problem to constraints w.r.t. the current inputs.

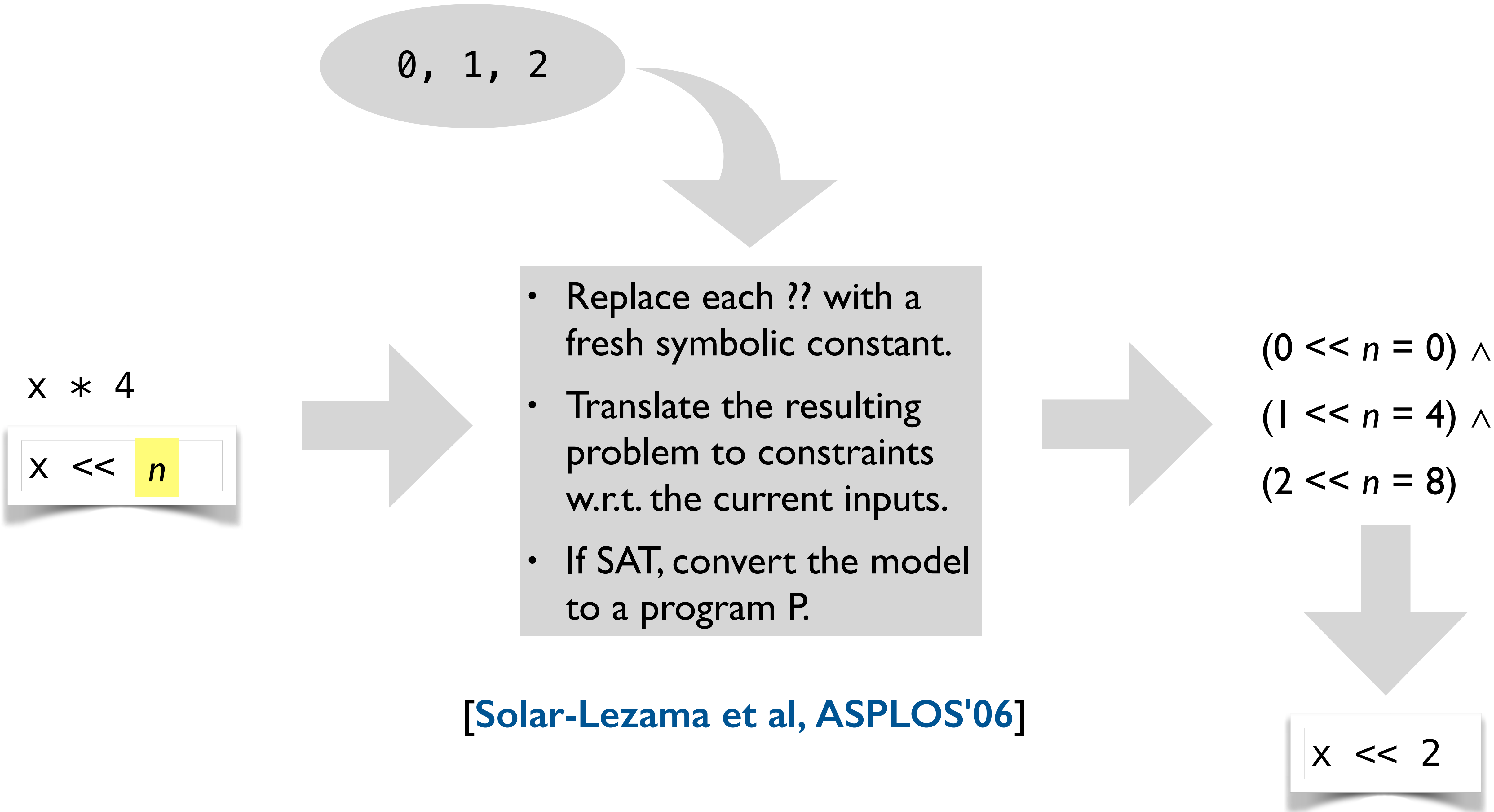
$(0 \ll n = 0) \wedge$

$(1 \ll n = 4) \wedge$

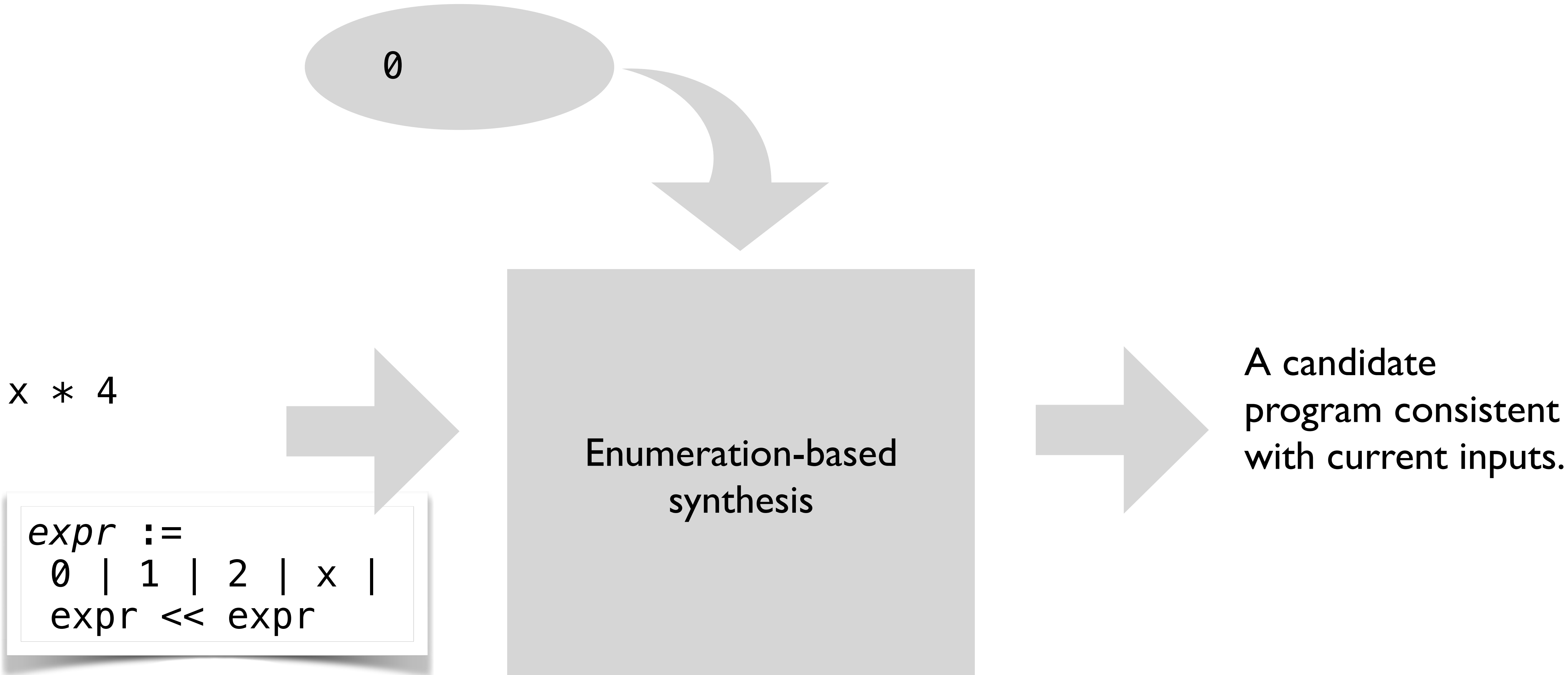
$(2 \ll n = 8)$

[Solar-Lezama et al, ASPLOS'06]

Synthesizing programs with a solver



Synthesizing programs with enumerative search



[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

```
0
```

```
x * 4
```

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

A candidate program consistent with current inputs.

[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

```
0
```

```
x * 4
```

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

K=*1*: 0

[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

```
0, 1
```

```
x * 4
```

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

0, 1

x * 4

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

0, 1

x * 4

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

K=*1*: 0, 1, 2, x

[Udupa et al, PLDI'13]

Synthesizing programs with enumerative search

`0, 1`

`x * 4`

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

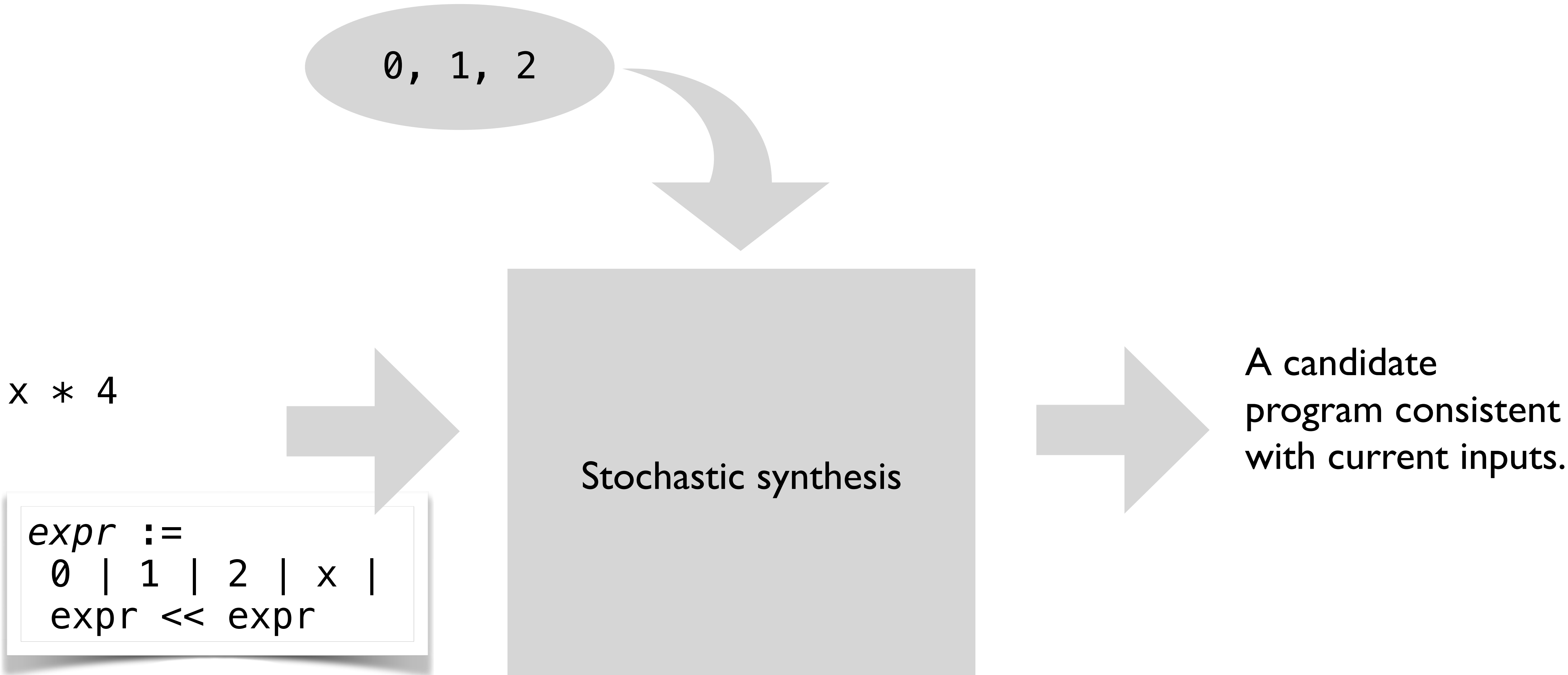
- Iteratively construct all programs of size *K* until one is consistent with the current inputs.
- If two programs produce the same output on all current inputs, keep just one of the two.

K=*1*: 0, 1, 2, x

K=2: 1 << 2, 2 << 2, x << 1, **x << 2**

[Udupa et al, PLDI'13]

Synthesizing programs with stochastic search



[Schkufza et al, ASPLOS'13]

Synthesizing programs with stochastic search

```
0, 1, 2
```

```
x * 4
```

```
expr :=
 0 | 1 | 2 | x |
 expr << expr
```

- Use Metropolis-Hastings to sample expressions.
- Mutate the current candidate program and keep the mutation with probability proportional to its correctness w.r.t. the current inputs.

A candidate program consistent with current inputs.

[Schkufza et al, ASPLOS'13]

Summary

Today

- Deductive and inductive synthesis
- Syntax-guided synthesis with symbolic, enumerative, and stochastic search

Next

- Two exciting guest lectures!
- Program verification in the real world.