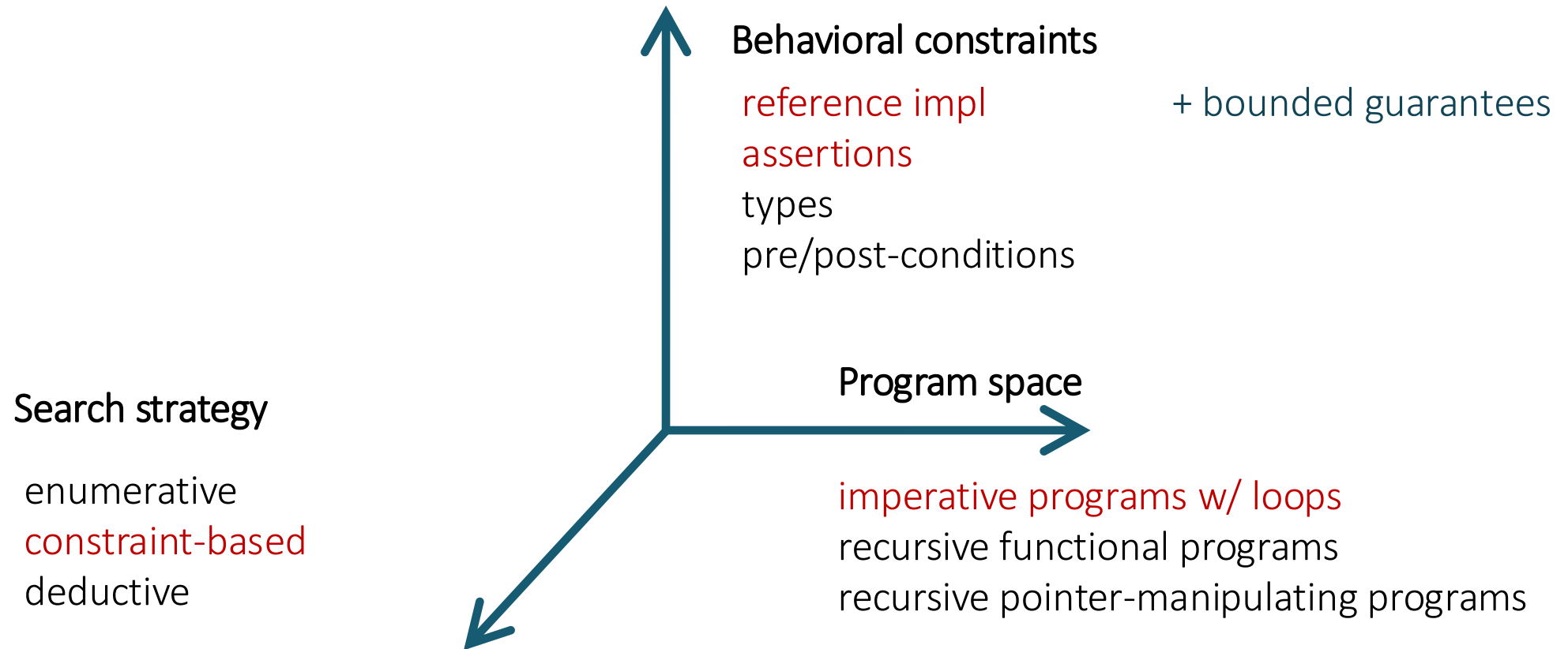


# Lecture 10

## Program Sketching

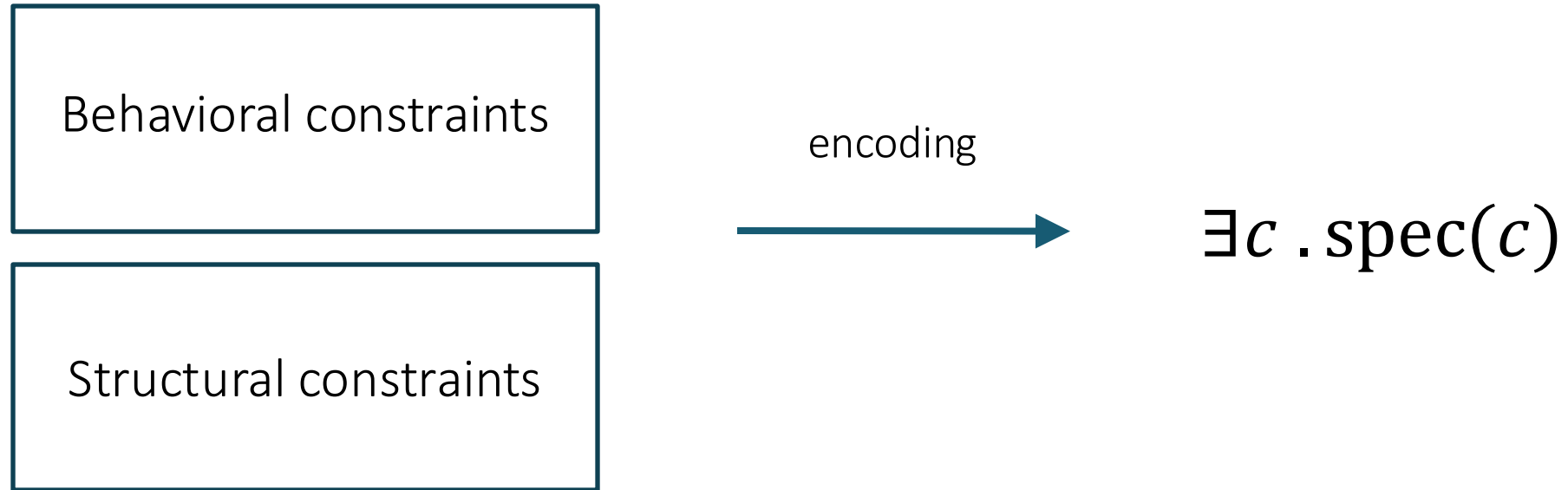
# Program Sketching

---



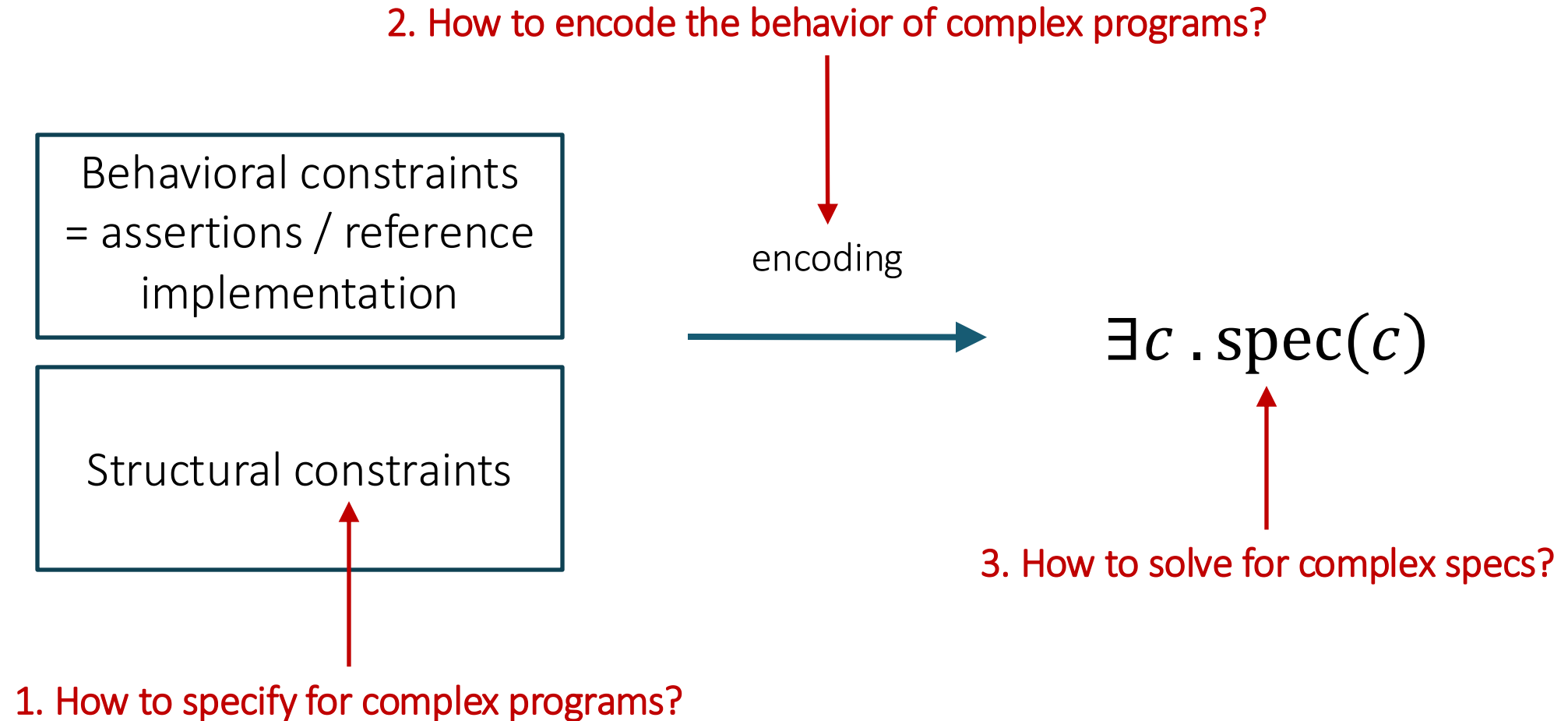
# Constraint-based synthesis

---



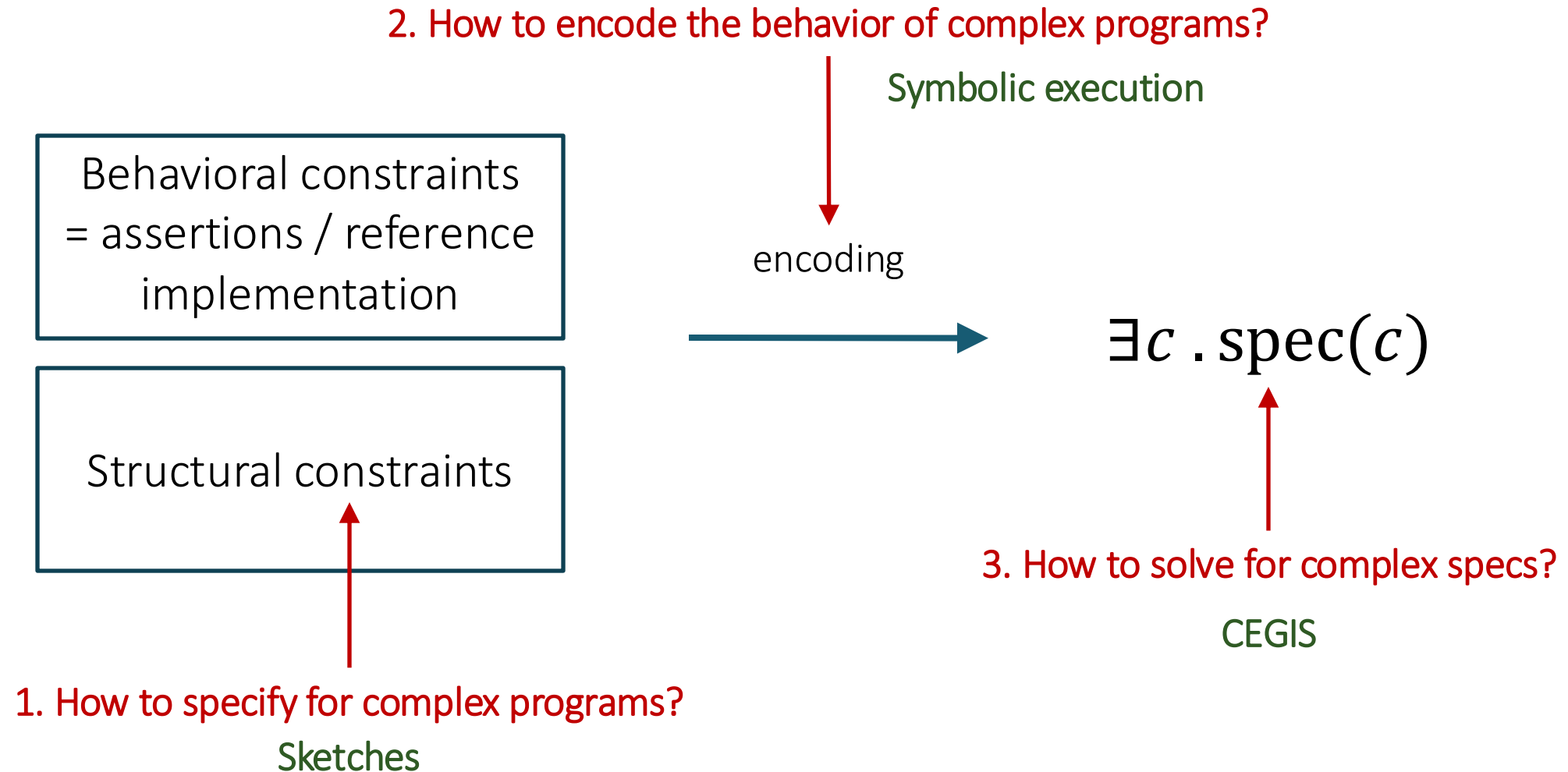
# Constraint-based synthesis for complex programs

---



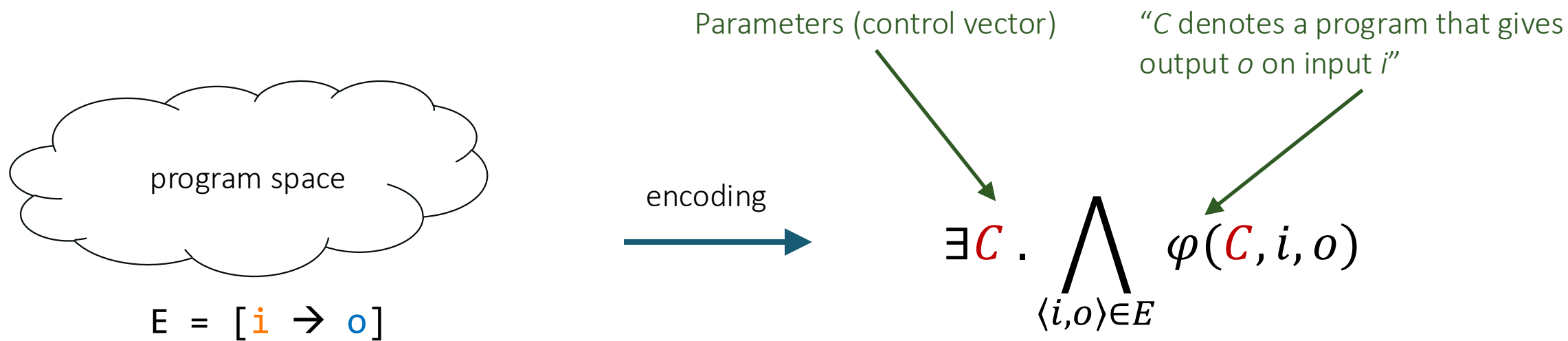
# Program Sketching

---

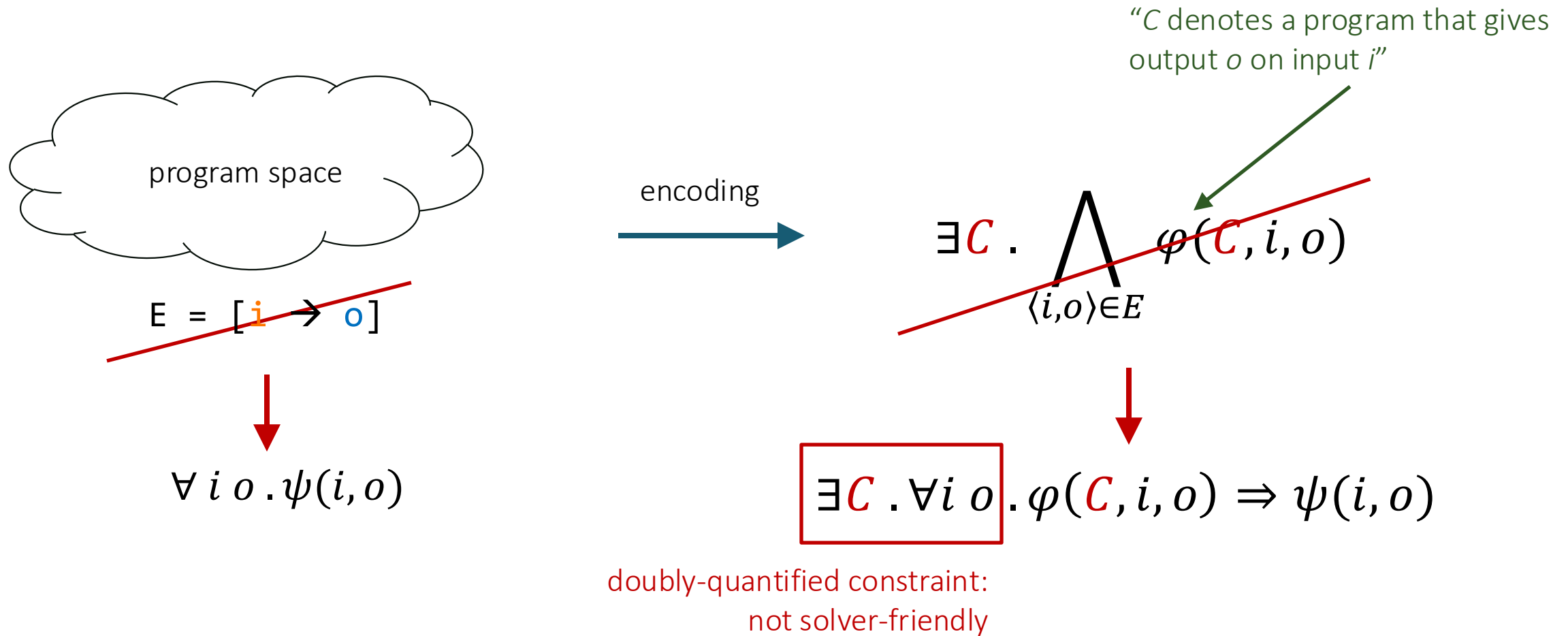


# Constraint-based synthesis from examples

---



# Constraint-based synthesis from specifications



# Example

---

```
harness void main(int x) {  
  int y := ?? * x + ??;  
  assert y - 1 == x + x;  
}
```

encoding



$$\exists \mathbf{C} . \forall i \ o . \varphi(\mathbf{C}, i, o) \Rightarrow \psi(i, o)$$

$$\begin{aligned} \exists c_1 c_2 . \forall x \ y . y &= c_1 * x + c_2 \\ &\Rightarrow y - 1 = x + x \end{aligned}$$

simplify



$$\exists c_1 c_2 . \forall x . c_1 * x + c_2 - 1 = x + x$$

How do we solve this constraint?



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

## Idea 1: Bounded Observation Hypothesis

- Assume there exists a small set of inputs  $X = \{x_1, x_2, \dots, x_n\}$  such that whenever  $c$  satisfies

$$\bigwedge_{i \in 1..n} Q(c, x_i)$$

it also satisfies

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

← No quantifiers here, can give to SAT / SMT

# Example

---

This is a linear constraint, two inputs are enough!

$$\exists c_1 c_2 . \forall x . c_1 * x + c_2 - 1 = x + x$$

$$X = \{0, 1\}$$

$$Q(c_1, c_2, 0) \equiv c_2 - 1 = 0$$

$$Q(c_1, c_2, 1) \equiv c_1 + c_2 - 1 = 2$$

$$\{c_1 \rightarrow 2, c_2 \rightarrow 1\}$$

```
harness void main(int x) {  
    int y := 2 * x + 1;  
    assert y - 1 == x + x;  
}
```

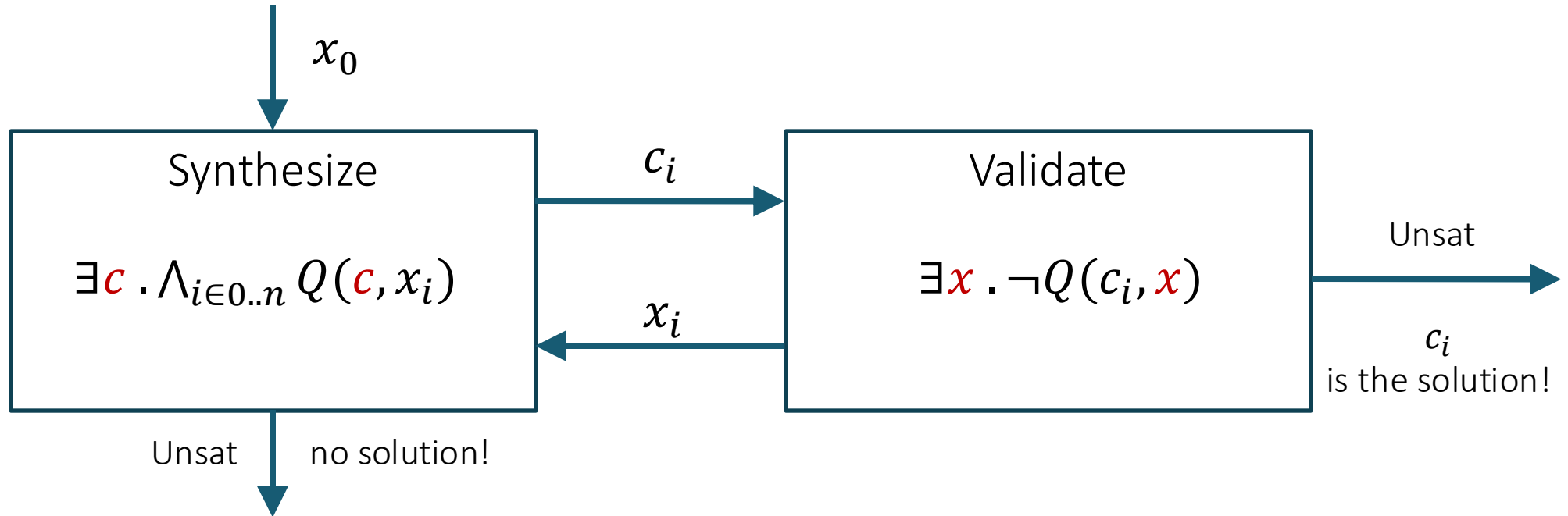
How do we find X in a general case?

# CEGIS

---

$$\exists \textcolor{red}{c} . \forall x . Q(\textcolor{red}{c}, x)$$

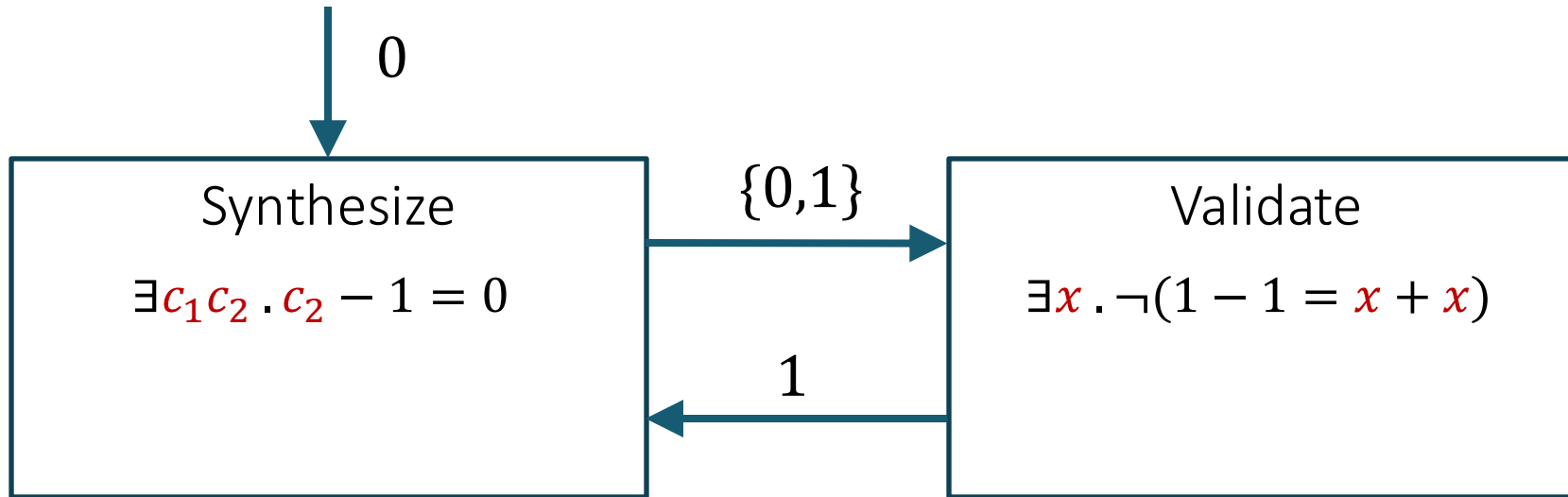
Idea 2: Rely on a validation oracle to generate counterexamples



# Example

---

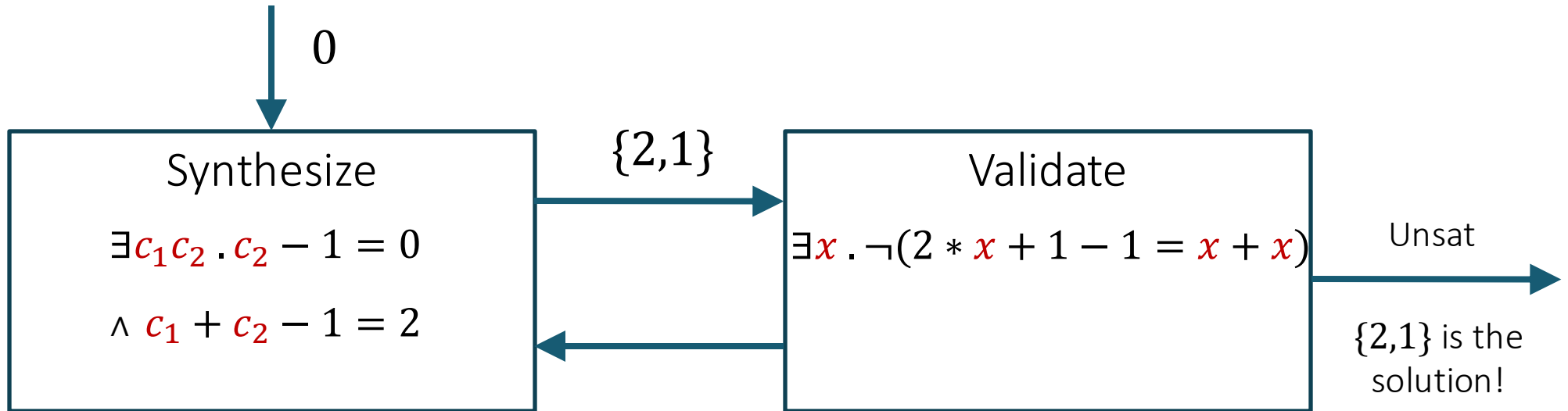
$$\exists c_1 c_2 . \forall x . c_1 * x + c_2 - 1 = x + x$$



# Example

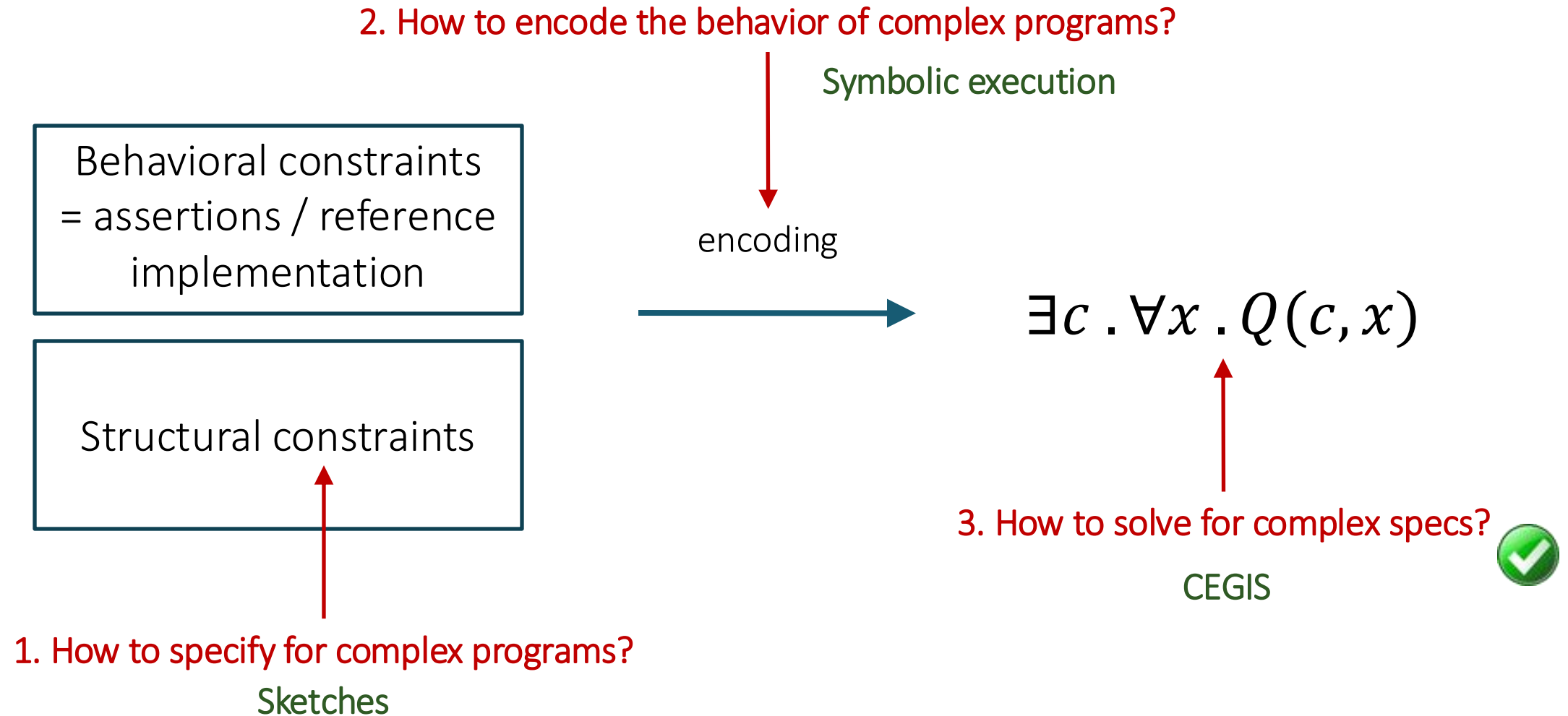
---

$$\exists c_1 c_2 . \forall x . c_1 * x + c_2 - 1 = x + x$$



# Program Sketching

---



# Structural constraints in Sketch

---

Different constraints good for different problems

- CFGs
- Components
- Just figure out the constants

**Idea:** Allow the programmer to encode all kinds of constraints using... programs (duh!)

# Language Design Strategy

---

Extend base language with one construct

Constant hole: ??

```
int bar (int x)
{
    int t = x * ??;
    assert t == x + x;
    return t;
}
```



```
int bar (int x)
{
    int t = x * 2;
    assert t == x + x;
    return t;
}
```

Synthesizer replaces ?? with a natural number



# Constant holes $\rightarrow$ sets of expressions

---

Expressions with  $??$  == sets of expressions

- linear expressions
- polynomials
- sets of variables

$x^{*}?? + y^{*}??$

$x^{*}x^{*}?? + x^{*}?? + ??$

$?? \ ? \ x : y$

# Example: swap without a temporary

---

Swap two integers without an extra temporary

```
void swap(ref int x, ref int y){  
    x = ... // sum or difference of x and y  
    y = ... // sum or difference of x and y  
    x = ... // sum or difference of x and y  
}
```

```
harness void main(int x, int y){  
    int tx = x; int ty = y;  
    swap(x, y);  
    assert x==ty && y == tx;  
}
```

# Syntactic sugar

---

`{ | RegExp | }`

RegExp supports choice '|' and optional '?'

- can be used arbitrarily within an expression
  - to select operands `{ | (x | y | z) + 1 | }`
  - to select operators `{ | x (+ | -) y | }`
  - to select fields `{ | n(.prev | .next)? | }`
  - to select arguments `{ | foo( x | y, z) | }`

Set must respect the type system

- all expressions in the set must type-check
- all must be of the same type

# Complex program spaces

---

**Idea:** To build complex program spaces from simple program spaces, borrow abstraction devices from programming languages

Function: abstracts expressions

Generator: abstracts set of expressions

- Like a function with holes...
- ...but different invocations → different code

# Example: swap without a temporary

---

```
generator int sign() {  
    if ?? {return 1;} else {return -1;}  
}
```

```
void swap(ref int x, ref int y){  
    x = x + sign()*y;           ➔ 1  
    y = x + sign()*y;           ➔ -1  
    x = x + sign()*y;           ➔ -1  
}
```

```
harness void main(int x, int y){  
    int tx = x; int ty = y;  
    swap(x, y);  
    assert x==ty && y == tx;  
}
```

# Recursive generators

---

Can generators encode a CFG?

$$\begin{array}{l} M ::= n \mid x * M \\ P ::= M \mid M + P \end{array}$$

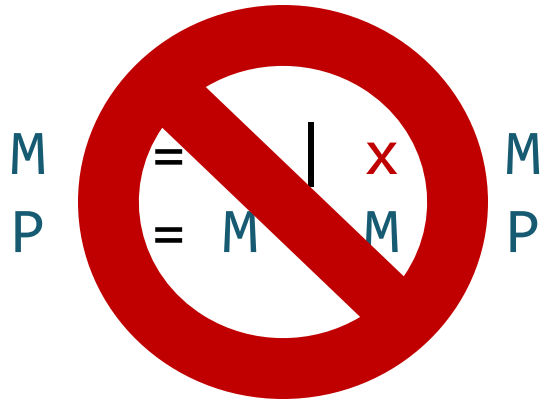
```
generator int mono(int x) {  
    if (??) {return ??;}  
    else {return x * mono(x);}  
}
```

```
generator int poly(int x) {  
    if (??) {return mono(x);}  
    else {return mono(x) + poly(x);}  
}
```

# Recursive generators

---

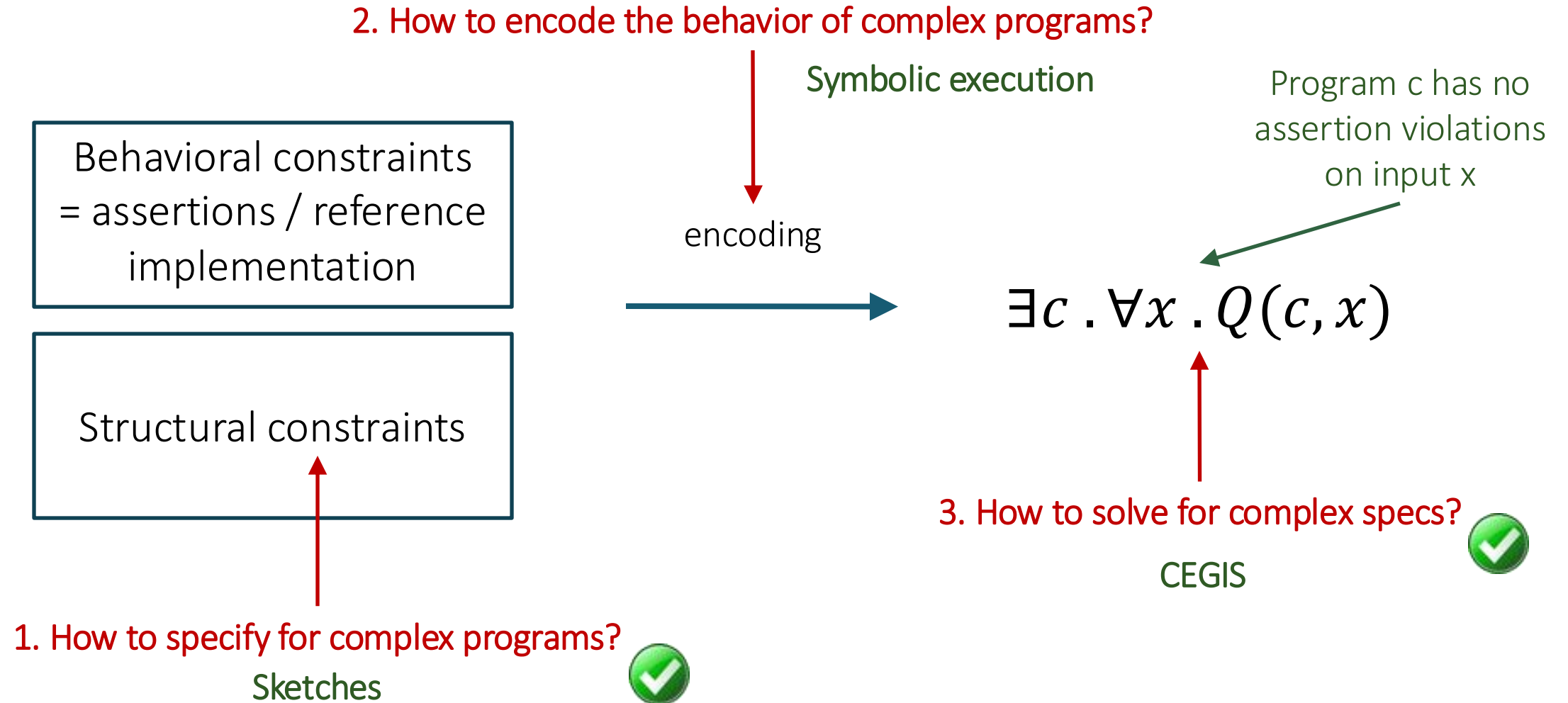
What if monomial of every degree can occur at most once?



```
generator int mono(int x, int n) {  
    if (n <= 0) {return ??;}  
    else {return x * mono(x, n - 1);}  
}
```

```
generator int poly(int x, int n) {  
    if (n <= 0) {return mono(x,0);}  
    else {return mono(x,n) + poly(x, n - 1);}  
}
```

# Program Sketching





# Symbolic execution

---

Semantics of a simple imperative language

How to use it for symbolic execution?

Adding while loops

Adding holes

# Semantics of a simple language

---

$e \quad := \quad n \mid x \mid e_1 + e_2$   
 $c \quad := \quad x := e \mid \mathbf{assert} \ e$   
 $\quad \mid c_1 ; c_2 \mid \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \mid \mathbf{while} \ e \ \mathbf{do} \ c$

What does an expression mean?

- An expression reads the state and produces a value
- The state is modeled as a map  $\sigma$  from variables to values
- $\mathcal{A}[\![\cdot]\!] : e \rightarrow \Sigma \rightarrow \mathbb{Z}$

Ex:

- $\mathcal{A}[\![x]\!] = \lambda\sigma. \sigma[x]$
- $\mathcal{A}[\![n]\!] = \lambda\sigma. n$
- $\mathcal{A}[\![e_1 + e_2]\!] = \lambda\sigma. \mathcal{A}[\![e_1]\!]\sigma + \mathcal{A}[\![e_2]\!]\sigma$

# Semantics of a simple language

---

$e \quad := \quad n \mid x \mid e_1 + e_2$   
 $c \quad := \quad x := e \mid \mathbf{assert} \ e$   
 $\quad \mid c_1 ; c_2 \mid \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \mid \mathbf{while} \ e \ \mathbf{do} \ c$

What does a command mean?

- A command modifies the state
- $\mathcal{C}[\cdot] : c \rightarrow \Sigma \rightarrow \Sigma$

Ex:

- $\mathcal{C}[x := e] = \lambda\sigma. \sigma[x \rightarrow (\mathcal{A}[e]\sigma)]$
- $\mathcal{C}[c_1; c_2] = \lambda\sigma. \mathcal{C}[c_2](\mathcal{C}[c_1]\sigma)$
- $\mathcal{C}[\mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2] = \lambda\sigma. \mathcal{A}[e]\sigma \ ? \ (\mathcal{C}[c_1]\sigma) : (\mathcal{C}[c_2]\sigma)$

# Semantics of assertions

---

$$\begin{aligned} e &:= n \mid x \mid e_1 + e_2 \\ c &:= x := e \mid \mathbf{assert} \ e \\ &\quad \mid c_1 ; c_2 \mid \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \mid \mathbf{while} \ e \ \mathbf{do} \ c \end{aligned}$$

What does a command mean?

- Commands also generate constraints on valid executions
- $\mathcal{C}[\![\cdot]\!] : c \rightarrow \langle \Sigma, \Psi \rangle \rightarrow \langle \Sigma, \Psi \rangle$

Constraints on values in initial  $\sigma$

Ex:

- $\mathcal{C}[\![\mathbf{assert} \ e]\!] = \lambda \langle \sigma, \psi \rangle. \langle \sigma, \psi \wedge \mathcal{A}[\![e]\!]\sigma \neq 0 \rangle$

# Symbolic execution

---

Semantics of a simple imperative language

How to use it for symbolic execution?

Adding while loops

Adding holes

# Concrete execution: example 1

---

Let's run this with  $x = 2$

```
void main(int x){  
    int y = 2 * x;  
    assert y > x;  
}
```

$$\begin{aligned}\sigma &= \{x \rightarrow 2\}, & \psi &= \top \\ \sigma &= \{x \rightarrow 2, y \rightarrow 4\}, & \psi &= \top \\ \sigma &= \{x \rightarrow 2, y \rightarrow 4\}, & \psi &= \{4 > 2\}\end{aligned}$$


Test passed

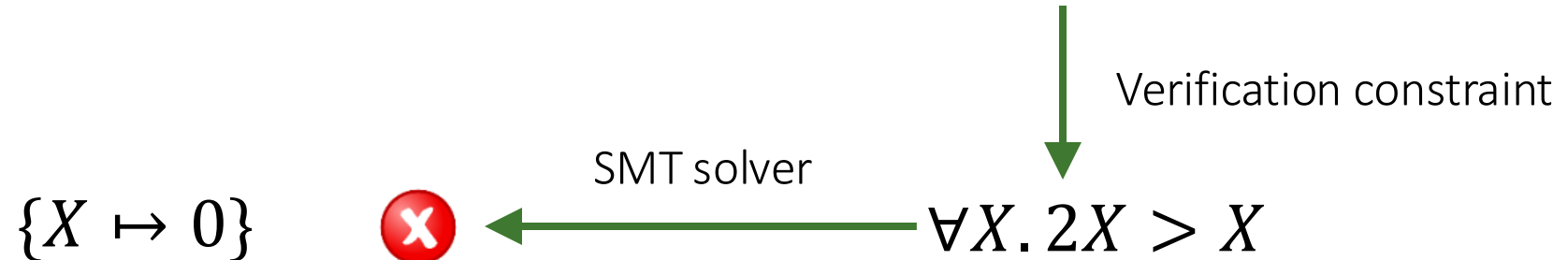
# Symbolic execution: example 1

---

```
void main(int x){  
  int y = 2 * x;  
  assert y > x;  
}
```

$\sigma = \{x \rightarrow X\}, \psi = \top$   
 $\sigma = \{x \rightarrow X, y \rightarrow 2X\}$   
 $\psi = \{2X > X\}$

$$\mathcal{C}[[p]]\langle\{\}, \top\rangle = \langle\{x \rightarrow X, y \rightarrow 2X\}, 2X > X\rangle$$



# Symbolic execution: example 2

```
→ void main(int x, int u){  
→   int y = 0;  
→   if (u > 0) {  
→     y = 2 * x;  
→   } else {  
→     y = x + x;  
→   }  
→   assert y == 2*x;  
→ }
```

$$\sigma = \{x \rightarrow X, u \rightarrow U\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, y \rightarrow 0\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, y \rightarrow 2X\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, y \rightarrow X + X\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, y \rightarrow U > 0 ? 2X : X + X\}$$

$$\psi = \{(U > 0 ? 2X : X + X) = 2X\} \quad \checkmark$$



# Symbolic execution

---

Semantics of a simple imperative language

How to use it for symbolic execution?

Adding while loops

Adding holes

# What about loops?

---

Semantics of a while loop

- Let  $W = \mathcal{C}[\textit{while } e \textit{ do } c]$
- $W$  satisfies the following equation:
$$W \sigma = \mathcal{A}[e] \sigma \text{ ? } (W(\mathcal{C}[c] \sigma)) : \sigma$$
- One strategy: find a fixpoint (see later in class)
- We'll settle for a simpler strategy: unroll k times and then give up

# Symbolic execution: example 3

---

```
void main(int x){  
  int y = 0;  
  int i = 0;  
  while (i < 2) {  
    y = y + x;  
    i = i + 1;  
  }  
  assert y == i * x;  
}
```

Step 1: unroll  
with depth = 2

```
if (i < 2) {  
  y = y + x;  
  i = i + 1;  
  if (i < 2) {  
    y = y + x;  
    i = i + 1;  
    assert !(i < 2);  
  }  
}
```

# Symbolic execution: example 3

```
→ void main(int x){  
  → int y = 0;  
  → int i = 0;  
  → if (i < 2) {  
    y = y + x;  
    i = i + 1;  
  →   if (i < 2) {  
    y = y + x;  
    i = i + 1;  
  →   assert !(i < 2);  
  →   }  
  → }  
  → assert y == i*x;  
}
```

$$\sigma = \{x \rightarrow X\}$$

$$\sigma = \{x \rightarrow X, y \rightarrow 0, i \rightarrow 0\}$$

$$\sigma = \{x \rightarrow X, y \rightarrow X, i \rightarrow 1\}$$

Simplified from  $0 < 2 ? (1 < 2 ? X + X : X) : 0$

$$\sigma = \{x \rightarrow X, y \rightarrow X + X, i \rightarrow 2\}$$

$$\psi = \{\neg(2 > 2)\}$$

$$\sigma = \{x \rightarrow X, y \rightarrow X + X, i \rightarrow 2\}$$

$$\psi = \{\neg(2 > 2) \wedge X + X = 2X\}$$



# Symbolic execution

---

Semantics of a simple imperative language

How to use it for symbolic execution?

Adding while loops

Adding holes

# Semantics of sketches

---

$$\begin{aligned} e &:= n \mid x \mid e_1 + e_2 \mid \textcolor{red}{??}_i \\ c &:= x := e \mid \textbf{assert } e \\ &\mid c_1 ; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \end{aligned}$$

What does an expression mean?

- Like before, but with a “hole environment”  $\phi$
- $\mathcal{A}[\cdot] : e \rightarrow \Phi \rightarrow \Sigma \rightarrow \mathbb{Z}$

Ex:

- $\mathcal{A}[x] = \lambda\phi. \lambda\sigma. \sigma[x]$
- $\mathcal{A}[??_i] = \lambda\phi. \lambda\sigma. \textcolor{red}{\phi}[i]$
- $\mathcal{A}[e_1 + e_2] = \lambda\phi. \lambda\sigma. \mathcal{A}[e_1]\phi\sigma + \mathcal{A}[e_2]\phi\sigma$

# Symbolic Evaluation of Commands

---

Commands have two roles

- Modify the symbolic state
- Generate constraints

$$\mathcal{C}[\![\cdot]\!] : c \rightarrow \Phi \rightarrow \langle \Sigma, \Psi \rangle \rightarrow (\Sigma, \Psi)$$

# Symbolic Evaluation of Commands

---

Example: assignment and assertion

$$\mathcal{C}[[x := e]]\phi \langle \sigma, \psi \rangle = \langle \sigma[x \mapsto \mathcal{A}[[e]]\phi\sigma], \psi \rangle$$

$$\mathcal{C}[[\text{assert } e]]\phi \langle \sigma, \psi \rangle = \langle \sigma, \psi \wedge \mathcal{A}[[e]]\phi\sigma \neq 0 \rangle$$



# Symbolic execution of sketches: example

```

→ void main(int x){
  int z = ??1 * x;
  int y = 0;
→  int i = 0;
  if (i < 2) {
    y = y + x;
    i = i + 1;
→    if (i < 2) {
      y = y + x;
      i = i + 1;
→    assert !(i < 2);
→  }
  }
  assert y == z;
}

```

$$\sigma = \{x \rightarrow X\} \quad \psi = \top$$

$$\sigma = \{x \rightarrow X, z \rightarrow \phi_1 * X, y \rightarrow 0, i \rightarrow 0\}$$

$$\sigma = \{x \rightarrow X, z \rightarrow \phi_1 * X, y \rightarrow X, i \rightarrow 1\}$$

$$\sigma = \{x \rightarrow X, z \rightarrow \phi_1 * X, y \rightarrow X + X, i \rightarrow 2\}$$

$$\psi = \{\neg(2 > 2)\}$$

$$\psi = \{\neg(2 > 2) \wedge X + X = \phi_1 * X\}$$

$$\{\phi_1 \mapsto 2\} \xleftarrow{\text{CEGIS}} \exists \phi_1. \forall X. X + X = \phi_1 * X$$

# Controls for generators

```
harness void main(int x, int y){  
  z = mono(x) + mono(y);  
  → assert z == x + x + 3;  
}
```

$$\sigma = \{z \rightarrow (\phi_1 ? \phi_2 : X * \phi_2) + (\phi_1 ? \phi_2 : Y * \phi_2)\}$$

No solution!

```
generator int mono(int x) {  
  if (??1) {return ??2;}  
  else {return x * mono(x);}  
}
```

unroll with  
depth = 1

```
if (??1) {return ??2;}  
else {return x * ??2;}  
}
```

We need to map different calls to mono to different controls!

# Controls for generators: context

---

```
harness void main(int x, int y){  
→   z = mono1(x, 1) + mono2(y, 2);  
   assert z == x + x + 3;  
}  
  
generator int mono(int x, context  $\tau$ ) {  
   if (?? $\tau_1$ ) {return ?? $\tau_2$ ;}  
   else {return x * mono3(x,  $\tau.3$ );}  
}
```

$$\sigma = \{z \rightarrow (\phi_1^1 ? \phi_2^1 : X * \phi_2^{1.3}) + (\phi_1^2 ? \phi_2^2 : X * \phi_2^{2.3})\}$$

$$\{\phi_1^1 \mapsto 0, \phi_2^{1.3} \mapsto 2, \phi_1^2 \mapsto 1, \phi_2^{1.3} \mapsto 3\}$$

# Sketch: contributions

---

Expressing structural and behavioral constraints as programs

- the only primitive extension is an integer hole ??
- why is it important to keep extensions minimal?

CEGIS

- became extremely popular; now used in most constraint-based synthesizers

Can discover constants

- like all constraint-based

# Sketch: limitations

---

Everything is bounded

- loops are unrolled
- integers are bounded
- are any of the above easily fixable?

Too much input from the programmer?

- but: as search gets better, less user input is required

CEGIS relies on the Bounded Observation Hypothesis

Sketches hard to debug

No bias, no non-functional constraints

# Sketch: questions

---

Behavioral constraints? structural constraints? search strategy?

- assertions / reference implementation
- sketches
- constraint-based (CEGIS + SAT)

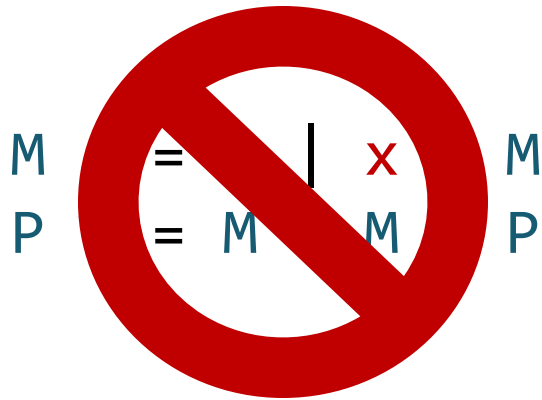
Sketches vs CFGs? Brahma's components?

- A generator can encode a multiset of components (although it's not very straightforward)
- Can a generator encode a CFG?

# Recursive generators

---

What if monomial of every degree can occur at most once?



```
generator int mono(int x, int n) {  
    if (n <= 0) {return ??;}  
    else {return x * mono(x, n - 1);}  
}
```

```
generator int poly(int x, int n) {  
    if (n <= 0) {return mono(x, 0);}  
    else {return mono(x, n) + poly(x, n - 1);}  
}
```

Generators are more expressive than CFGs!

- but unbounded generators cannot be encoded into constraints
- need to bound unrolling depth
- bounded generators less expressive than CFGs (but more convenient)

# Semantics of abort

---

$$\mathcal{C}[\text{abort}]\langle\sigma, \psi\rangle = \langle\sigma, \perp\rangle$$



# CEGIS: the worst case

---

Satisfiable constraint  $\exists c. \forall x. Q(c, x)$  that violates the Bounded Observation Hypothesis

$$Q(c, x) \equiv x \neq c \quad \text{unsatisfiable}$$

$$Q(c, x) \equiv c = (x \text{ XOR } x) \quad \text{solved in single iteration for ANY } x$$

$$Q(c, x) \equiv x \leq c \quad \begin{array}{l} \text{solved in one iteration with } x = 111 \\ \text{(but will require } 2^N \text{ iterations with worst-case counterexamples)} \end{array}$$

$$Q(c, x) \equiv x = (x \& c) \quad \text{solved in max } n \text{ iterations}$$

$$Q(c, x) \equiv c \neq x \vee c = 0 \quad \begin{array}{l} \text{violates BOH:} \\ \text{no small set of counter-examples exists} \end{array}$$