CS5733 Program Synthesis

#13.Sketching and constraints based search

Ashish Mishra, September 20, 2024

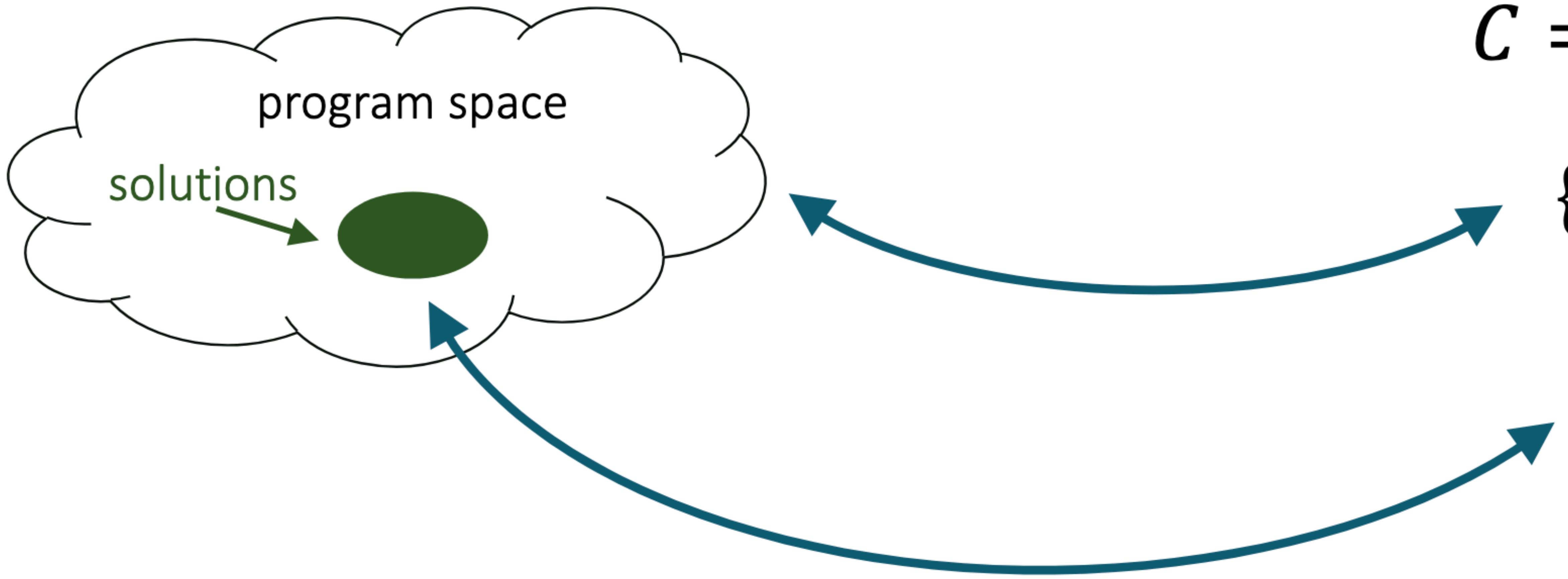
RECAP

- CBS
  - Parameterized Programs
- Sketches : A language to write Parameterized Programs
- Language
  - Imperative language
  - Unknown Constants
  - Harness/ Test-harness
  - Generators : Allow complex parametrized programs
    - Recursive
    - Higher-order

Constraint-Based Search

Idea: encode the synthesis problem as a SAT/SMT problem and let a solver deal with it

What is an Encoding



parameter space

$$C = \{c_1, \dots, c_n\}$$

$$\{C \mid \mathrm{wf}(C)\}$$

well-formedness constraint

$$\{C \mid \mathrm{wf}(C) \wedge \mathrm{sem}(C)\}$$

semantic correctness constraint

$$\mathrm{sem}(C) = \bigwedge_{\langle i,o \rangle \in E} \varphi(C, i, o)$$

C denotes a program that gives output *o* on input *i*

How to define an encoding

Define the parameter space $C = \{c_1, \ldots, c_n\}$

- `decode : C → Prog` (might not be defined for all C)

Define a formula $\mathrm{wf}(c_1, \ldots, c_n)$

- that holds iff `decode[C]` is defined

Define a formula $\varphi(c_1, \ldots, c_n, i, o)$

- that holds iff `(decode[C])(i) = o`

Solve the constraints

```
constraint-based (wf, φ, E = [i → o]) {
  match SAT(wf(C) ∧ ⋀⟨i,o⟩∈E φ(C, i, o)) with
    Unsat -> return “No solution”
    Model C* -> return decode[C*]
}
```

Find a satisfying assignment for $c_1, \ldots, c_n$ (**i** and **o** are fixed)

SAT encoding: example

program space

`{x, x & 1}`

x is a two-bit word
(`x = x`$_h$`x`$_l$)

`E = [11 → 01]`

parameter space

$$C = \{c\text{: Bool}\}$$

```
decode[0] → x
decode[1] → x & 1
```

$$\text{wf}(c) \equiv \top$$

$$\varphi(c, i_h, i_l, o_h, o_l) \equiv (\neg c \Rightarrow o_h = i_h \wedge o_l = i_l)$$
$$\wedge\ (c \Rightarrow o_h = 0 \wedge o_l = i_l)$$

$$\text{SAT}(\varphi(c, 1, 1, 0, 1))$$

$$\text{SAT}((\neg c \Rightarrow 0 = 1 \wedge 1 = 1) \wedge (c \Rightarrow 0 = 0 \wedge 1 = 1))$$

SAT solver →

`Model {c→1}`

return `decode[1]` i.e. `x & 1`

SMT encoding: example

program space

`x + N | x * N`

N is an in integer literal
x is an integer input

`E = [2 → 9]`

parameter space

$$C = \{c_{op}: \text{Bool}, c_N: \text{Int}\}$$

```
decode[0,N] → x + N
decode[1,N] → x * N
```

$$\text{wf}(c_{op}, c_N) \equiv \top$$

$$\varphi(c_{op}, c_N, i, o) \equiv (\neg c_{op} \Rightarrow o = i + c_N) \wedge (c_{op} \Rightarrow o = i * c_N)$$

$$\text{SAT}(\varphi(c_{op}, c_N, 2,9))$$

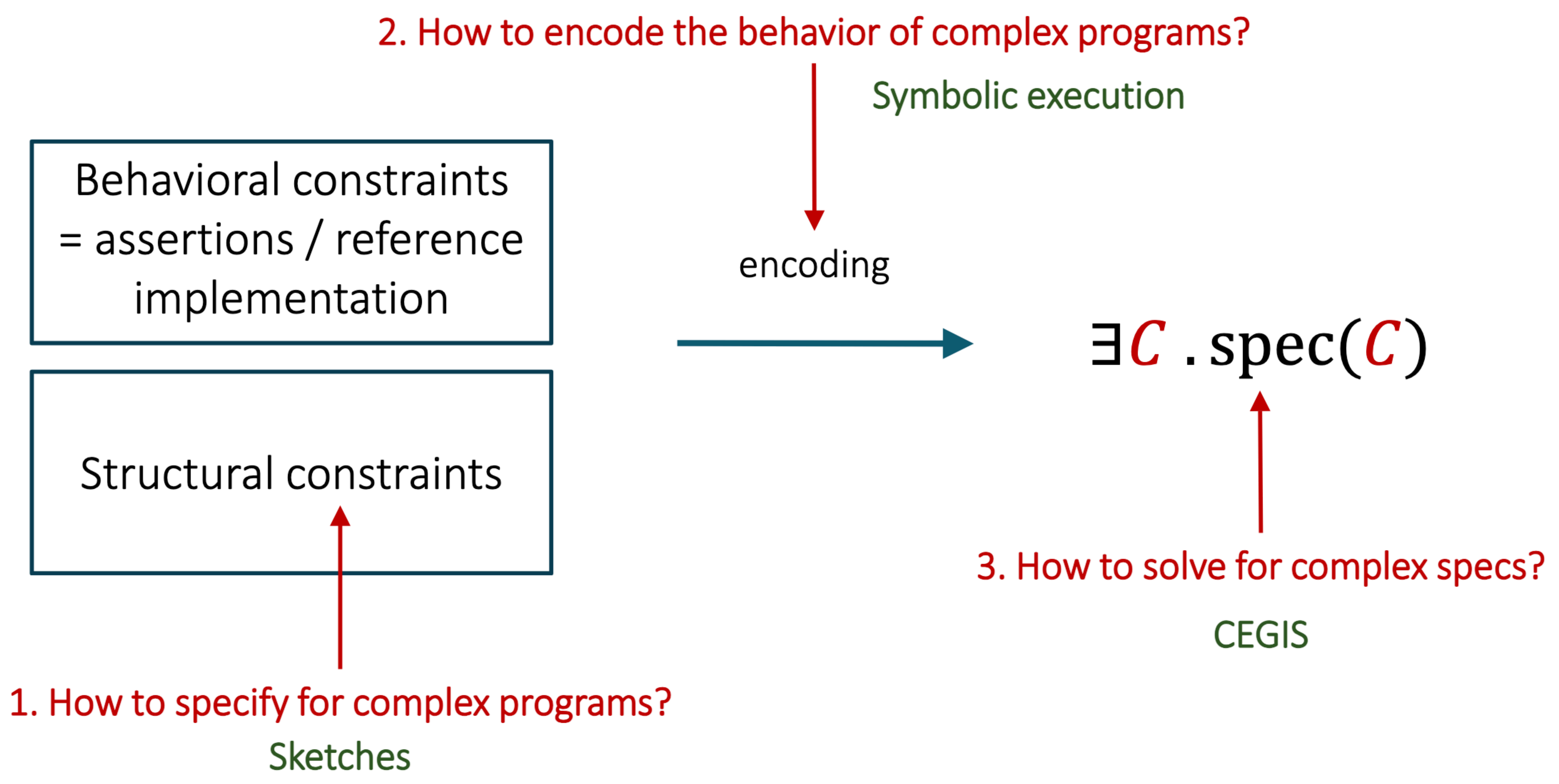
$$\text{SAT}((\neg c_{op} \Rightarrow 9 = 2 + c_N) \wedge (c_{op} \Rightarrow 9 = 2 * c_N))$$

SMT solver →

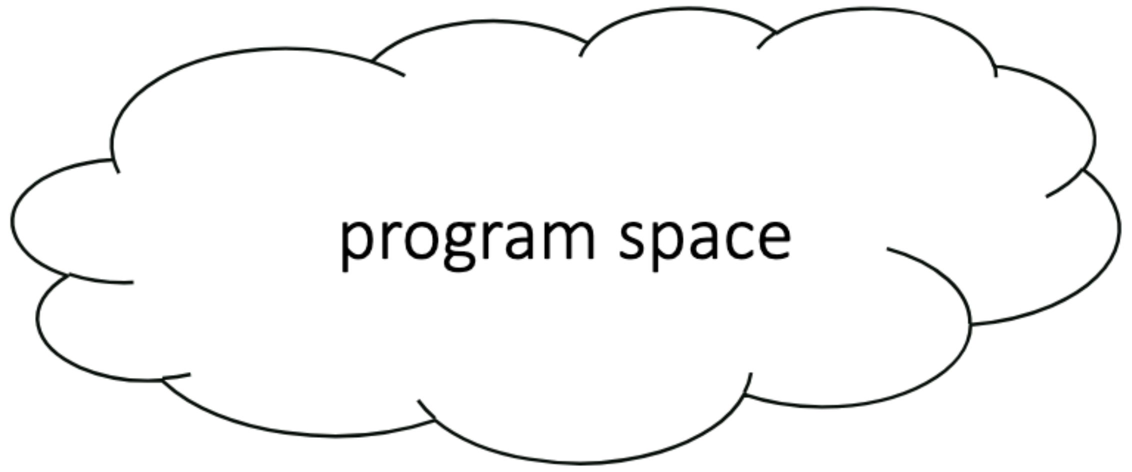
Model {c_{op}→0, c_N→7}

`return decode[0,7]` i.e. `x + 7`

Program Sketching



A small change: specs rather than i/o



~~E = [i → o]~~

↓

$\forall\, i\; o\, .\, \psi(i, o)$

encoding →

"*C* denotes a program that gives output *o* on input *i*"

~~$\exists C\, .\, \bigwedge_{\langle i,o \rangle \in E} \varphi(C, i, o)$~~

↓

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

doubly-quantified constraint:
not solver-friendly

Example : Sketches

$$\exists C\,.\,\forall i\;o\,.\,\varphi(C, i, o) \Rightarrow \psi(i, o)$$

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

encoding →

$$\exists c_1 c_2\,.\,\forall x\;y\,.\,y = c_1 * x + c_2 \\ \Rightarrow y - 1 = x + x$$

simplify ↓

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

Constraint-based search Over sketched Space

Overall Strategy

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:= n | x | $e_1 + e_2$ | $e_1 > e_2$

c:= x := e | c_1 ; c_2 | if e then c_1 else c_2 | while e do c

What does an expression mean?

- An expression reads the state and produces a value
- The state is modeled as a map σ from vars to values
- $\mathscr{A}[\![\cdot]\!] : e \to \Sigma \to int$

Ex:

- $\mathscr{A}[\![x]\!] = \lambda\sigma.\ \sigma(x)$
- $\mathscr{A}[\![n]\!] = \lambda\sigma.\ n$
- $\mathscr{A}[\![e_1 + e_2]\!] = \lambda\sigma.\ \mathscr{A}[\![e_1]\!]\sigma + \mathscr{A}[\![e_2]\!]\sigma$
- $\mathscr{A}[\![e_1 > e_2]\!] = \lambda\sigma.\ if\ \mathscr{A}[\![e_1]\!]\sigma > \mathscr{A}[\![e_2]\!]\sigma\ then\ 1\ else\ 0$

Semantics of a simple language

e:= n | x | e_1 + e_2

c:= x := e | c_1 ; c_2 | if e then c_1 else c_2 | while e do c

What does a command mean?

- A command modifies the state
- $\mathscr{C}[\![\cdot]\!] : c \to \Sigma \to \Sigma$

Ex:

- $\mathscr{C}[\![x := e]\!] = \lambda\sigma.\ \sigma[x \to (\mathscr{A}[\![e]\!]\sigma)]$
- $\mathscr{C}[\![c_1; c_2]\!] = \lambda\sigma.\mathscr{C}[\![c_2]\!](\mathscr{C}[\![c_1]\!]\sigma)$
- $\mathscr{C}[\![if\ e\ then\ c_1 else\ c_2]\!] =$

$$\lambda\sigma.\lambda x.\ if \mathscr{A}[\![e]\!]\sigma = 1\ then \left(\mathscr{C}[\![c_1]\!]\sigma\right)x\ else\left(\mathscr{C}[\![c_2]\!]\sigma\right)x$$

Semantics of assertions

```
e   :=  n | x | e₁ + e₂
c   :=  x := e | assert e
      | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

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 σ

Ex:

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

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;
}
```

$\sigma = \{x \to 2\}, \quad \psi = \top$

$\sigma = \{x \to 2, y \to 4\}, \psi = \top$

$\sigma = \{x \to 2, y \to 4\}, \psi = \{4 > 2\}$

Test passed

Symbolic execution : Example 1

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

$\sigma = \{x \to X\}, \psi = \top$

$\sigma = \{x \to X, y \to 2X\}$

$\psi = \{\, 2X > X\}$

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

Verification constraint

$$\forall X. 2X > X$$

SMT solver

$$\{X \mapsto 0\}$$

https://madhu.cs.illinois.edu/cs598-fall10/king76symbolicexecution.pdf

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;
}
```

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

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

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

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

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

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

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 $W : \Sigma \rightarrow \Sigma = \mathcal{C}[\![while\ e\ do\ c]\!]$
- W satisfies the following equation:

$$W\ \sigma \quad = \quad \mathcal{A}[\![e]\!]\sigma \neq 0 \ \ ? \ \ W(\mathcal{C}[\![c]\!]\sigma) \ : \ \sigma$$

- 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

```c
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

```c
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;
}
```

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

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

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

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

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

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

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

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

Symbolic Execution

Semantics of a simple imperative language

How to use it for symbolic execution?

Adding while loops

Adding holes

Semantics of sketches

```
e  :=  n | x | e₁ + e₂ | ??ᵢ
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

What does an expression mean?

- Like before, but with a "hole environment" ϕ
- $\mathcal{A}[\![\cdot]\!] : e \to \mathbf{\Phi} \to \Sigma \to \mathbb{Z}$

$\phi : H \to \mathbb{Z}$

Ex:

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

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

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

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

Symbolic Evaluation of Commands

If statement

$$\mathcal{C}[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]^{\tau}\langle \sigma , \Phi \rangle = \langle \sigma' , \Phi' \rangle$$

$$\begin{aligned}
\Phi_t &= \{\phi \in \Phi : \ \mathcal{A}[\![e]\!]^{\tau}\sigma\phi = true\} \\
\Phi_f &= \{\phi \in \Phi : \ \mathcal{A}[\![e]\!]^{\tau}\sigma\phi = false\} \\
\langle \sigma_1 , \Phi_1 \rangle &= \ \mathcal{C}[\![c_1]\!]^{\tau}\langle \sigma , \Phi_t \rangle \\
\langle \sigma_2 , \Phi_2 \rangle &= \ \mathcal{C}[\![c_2]\!]^{\tau}\langle \sigma , \Phi_f \rangle \\
\Phi' &= (\Phi_1) \cup (\Phi_2) \\
\sigma' &= \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^{\tau}\sigma\phi \ ?\ \sigma_1 x\phi \ :\ \sigma_2 x\phi
\end{aligned}$$

Symbolic execution of sketch : 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;
}
```

$\sigma = \{x \to X\}$ $\psi = \top$

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

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

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

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

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

$\exists \phi_1 . \forall X . X + X = \phi_1 * X$

CEGIS

$\{\phi_1 \mapsto 2\}$

Controls for generators

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

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

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

No solution!

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 τ) {
    if (??τ₁) {return ??τ₂;}
    else {return x * mono³(x, τ.3);}
}
```

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

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

Example

- Goal: Find a function from holes to values
  - Easy in the absence of generators

```
bit[W] isolateSk (bit[W] x) implements isolate0 {
        return !(x + ϕ(??1)) & (x + ϕ(??2)) ;
}
```

  - Finite set of holes so function is just a table

Framing the synthesis problem

- Generators with recursion need something more

```
generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??1) return x;
    if(??2) return ??5;
    if(??3) return ~gen_g1(x, bnd-1);
    if(??4){
        ...
    }
}

bit[W] isolate0sk (bit[W] x)  implements isolate0 {
     return gen_g0(x, 3);
}
```

because the same syntactic instance of a hole is supposed to have different values for different instances of the generator.

Framing the synthesis problem

- Generators need something more

```
generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(φ(??1)) return x;
    if(φ(??2)) return φ(??5);
    if(φ(??3)) return ~gen_g1(x, bnd-1);
    if(φ(??4)){
        ...
    }
}

bit[W] isolate0sk (bit[W] x)  implements isolate0 {
     return gen_g0(x, 3);
}
```

Make ϕ a function of a hole

and a Context

Framing the synthesis problem

- Generators need something more
  - The value of the holes depends on the context

```
generator bit[W] gen(context τ, bit[W] x, int bnd){
    assert bnd > 0;
    if(ϕ(τ,??1)) return x;
    if(ϕ(τ,??2)) return ϕ(τ,??5);
    if(ϕ(τ,??3)) return ~gen_g1(τ·g1, x, bnd-1);
    if(ϕ(τ,??4)){
        ...
    }
}

bit[W] isolate0sk (bit[W] x)  implements isolate0 {
     return gen_g0(g0, x, 3);
}
```

Solving Constraints: CEGIS

CEGIS

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

Idea 1: Bounded Observation Hypothesis

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

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

No quantifiers here, can give to SAT / SMT

it also satisfies

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

Example

This is a linear constraint, two inputs are enough!

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

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

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

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

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

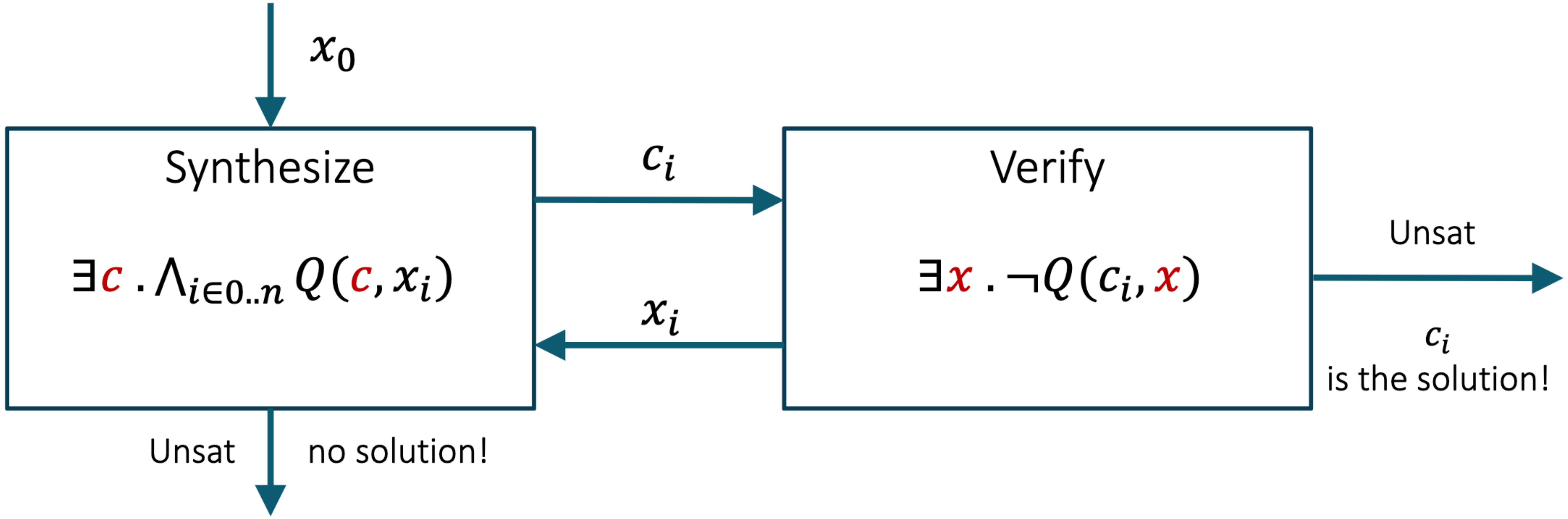
```
harness void main(int x) {
  int y := 2 * x + 1;
  assert y - 1 == x + x;
}
```

How do we find X in a general case?

CEGIS

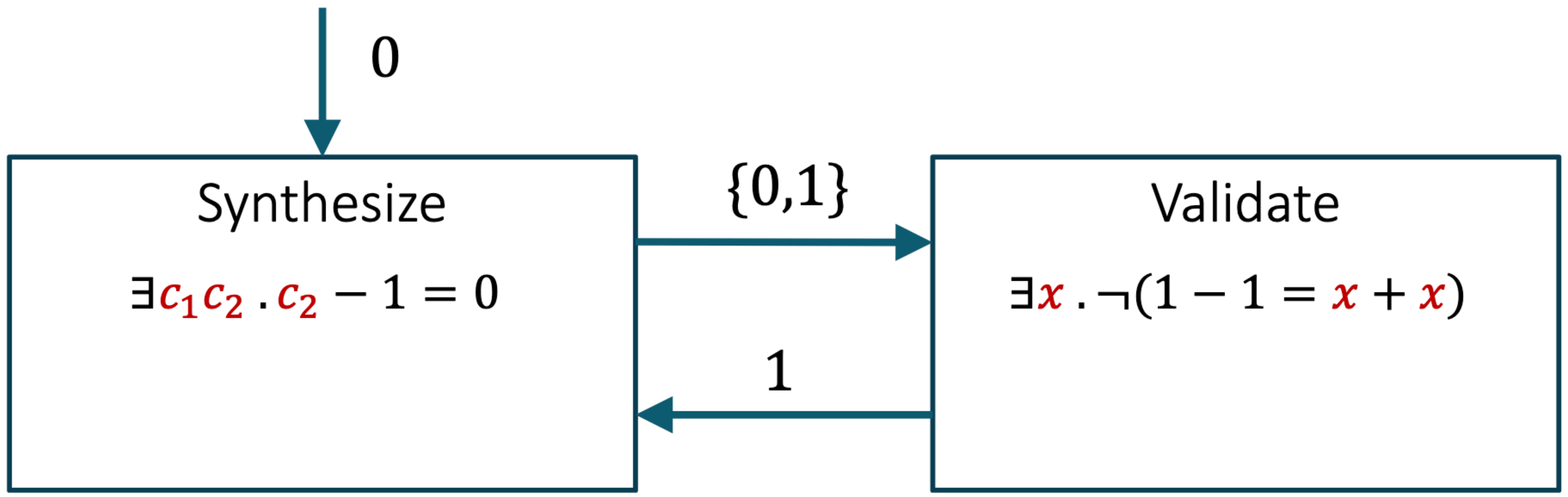
$$\exists c\,.\,\forall x\,.\,Q(c,x)$$

Idea 2: Rely on verification oracle to generate counterexamples



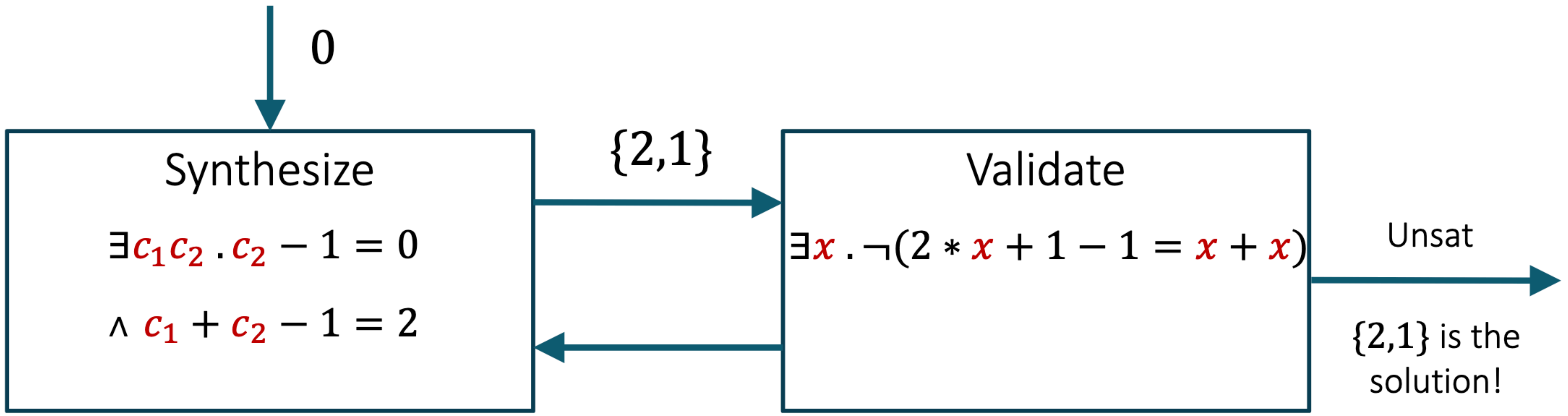
Example

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

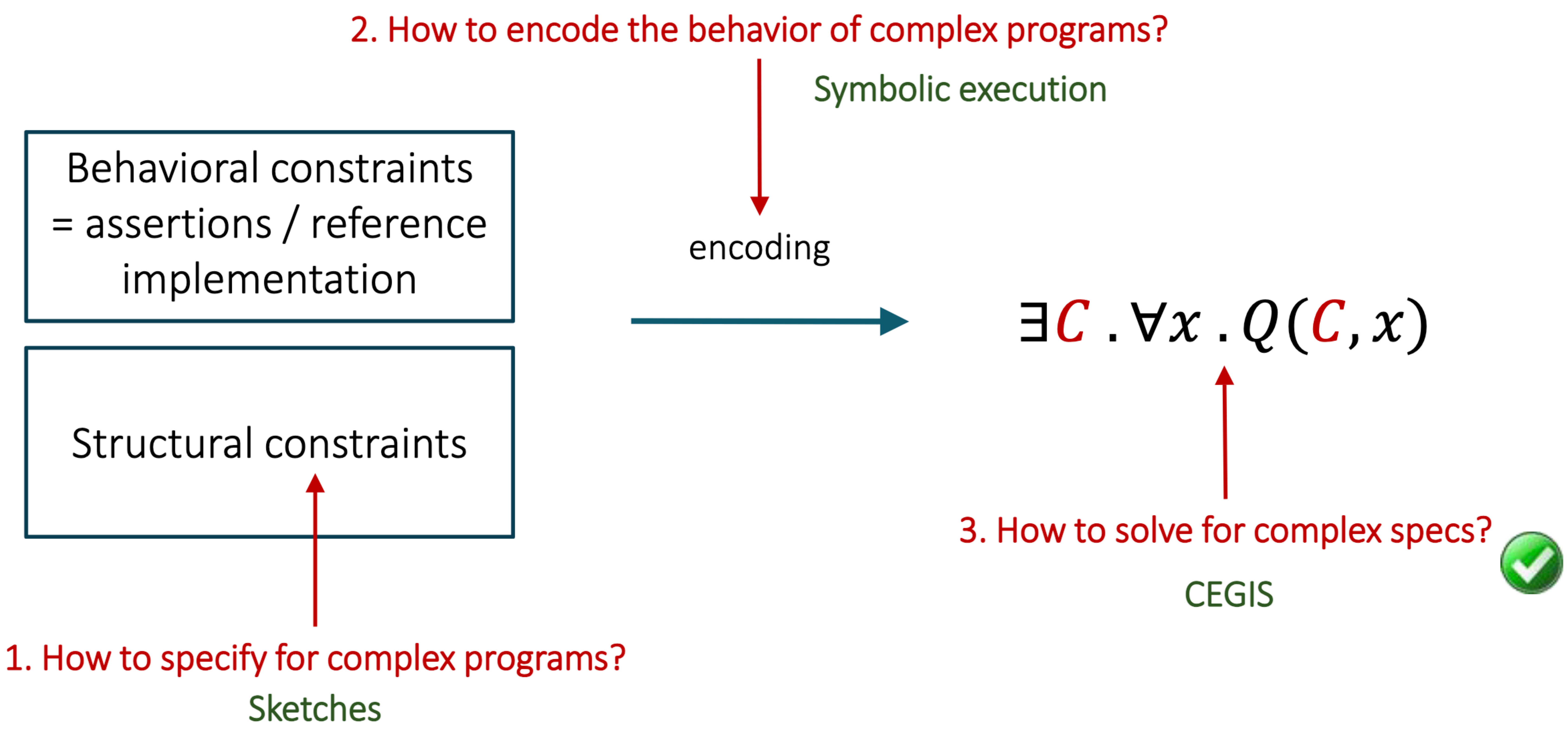


Example

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



Program Sketching



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?

Synthesis by translating to SAT

CEGIS

- became extremely popular!

Handles imperative programs with loops

- and proposes an encoding for those

Can discover constants

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

Logistics

- Reading assignment due on Sunday.
- The 1-2 page proposal, due on next Friday.
  - Something similar to the Sec. 1 and 2 in the papers.
  - A concrete example showing inputs to the synthesizer.
  - How does algorithm roughly works on the example.
- A class test 10 marks:
  - October 1, Tuesday
  - Syllabus, TBA