Lecture 10
Program Sketching

Nadia Polikarpova

Program Sketching

Behavioral constraints

reference impl

assertions

types

pre/post-conditions

\+ bounded guarantees

Program space

imperative programs w/ loops

recursive functional programs

recursive pointer-manipulating programs

Search strategy

enumerative

constraint-based

deductive

Constraint-based synthesis

Behavioral constraints

Structural constraints

encoding

$$\exists c \, . \, \mathrm{spec}(c)$$

CBS for complex programs

2. How to encode the behavior of complex programs?

Behavioral constraints
= assertions / reference
implementation

Structural constraints

encoding

$$\exists c\,.\,\mathrm{spec}(c)$$

3. How to solve for complex specs?

1. How to specify for complex programs?

Program Sketching

2. How to encode the behavior of complex programs?

Symbolic execution

Behavioral constraints
= assertions / reference implementation

Structural constraints

encoding

$$\exists c \,.\, \mathrm{spec}(c)$$

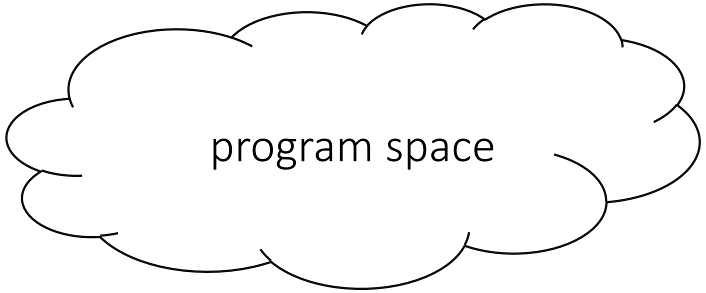
3. How to solve for complex specs?

CEGIS

1. How to specify for complex programs?

Sketches

CBS from examples



E = [i → o]

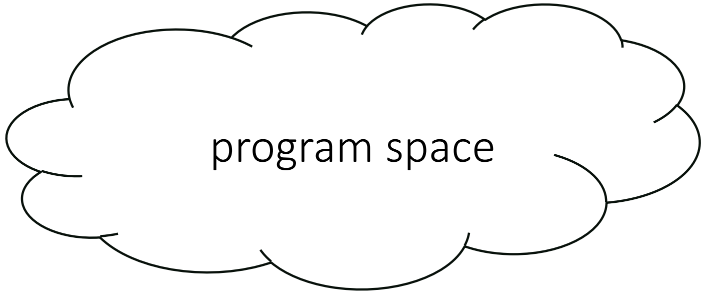
encoding →

Parameters (control vector)

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

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

CBS from specifications



~~E = [i → o]~~

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

encoding

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

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

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

doubly-quantified constraint:
not solver-friendly

Example

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

encoding $\longrightarrow$

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

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

simplify $\downarrow$

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

How do we solve this constraint?

CEGIS

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

Idea 1: Bounded Observation Hypothesis

- Assume there exists a small set of inputs $\mathrm{X} = \{x_1, x_2, \ldots x_n\}$ 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!

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

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

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

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

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

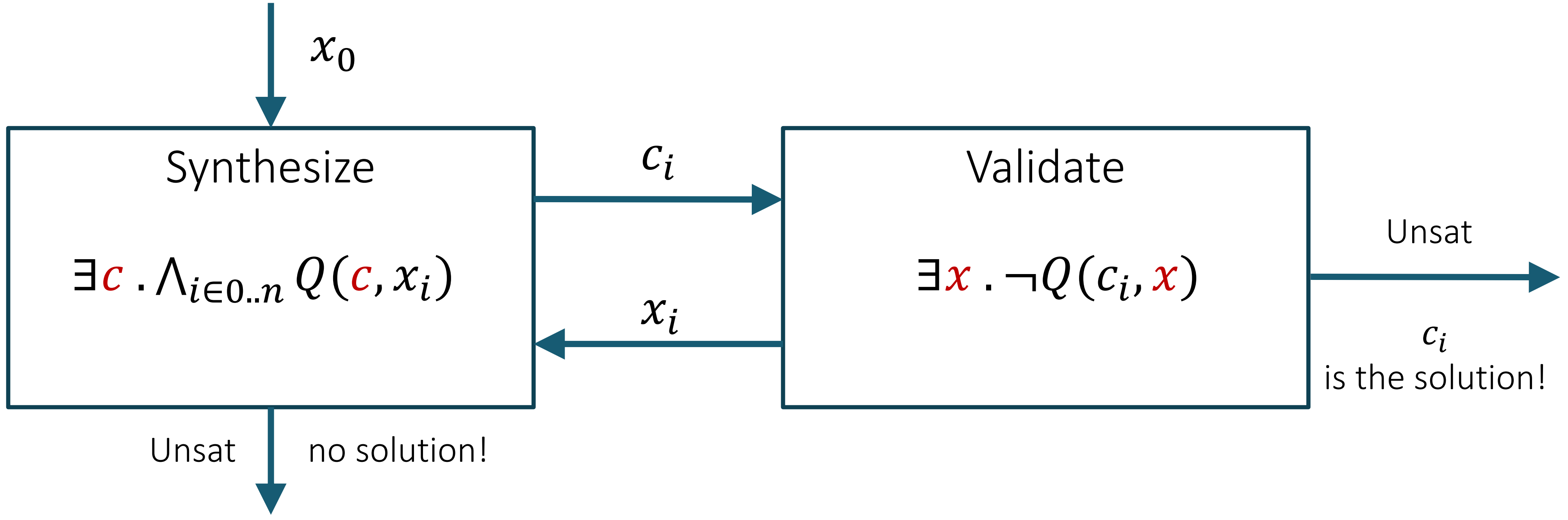
```
harness void main(int x) {
  int y := 2 * x + 1;
  assert y - 1 == x + x;
}
```

How do we find X in a general case?

CEGIS

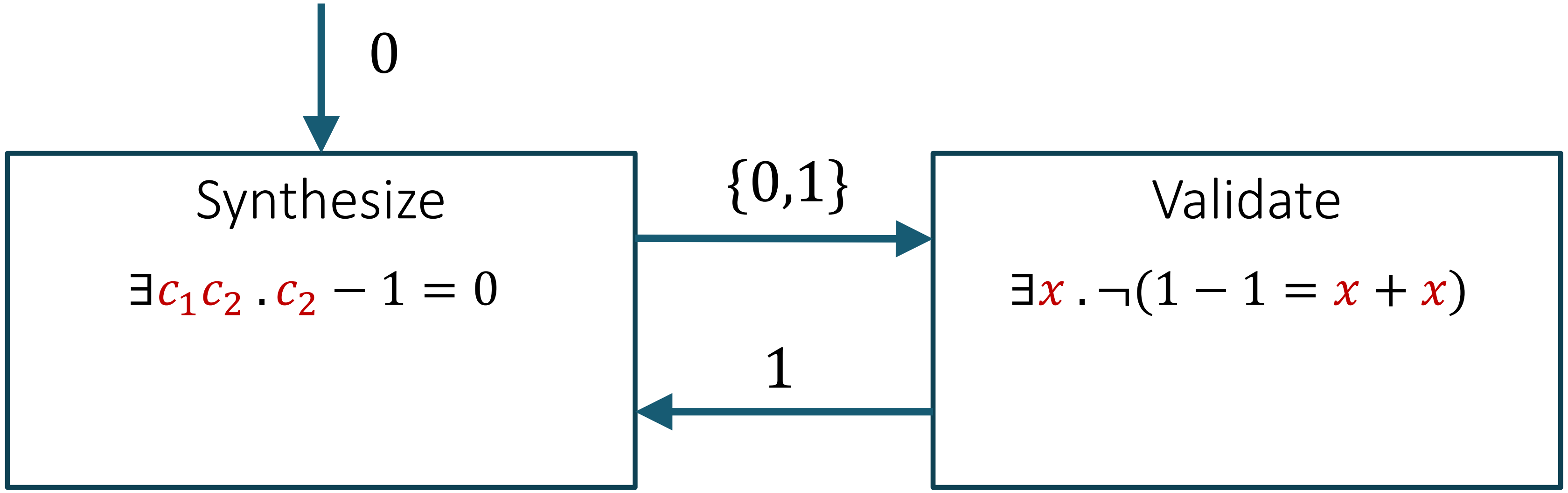
$$\exists c\,.\,\forall x\,.\,Q(c, x)$$

Idea 2: Rely on a validation oracle to generate counterexamples



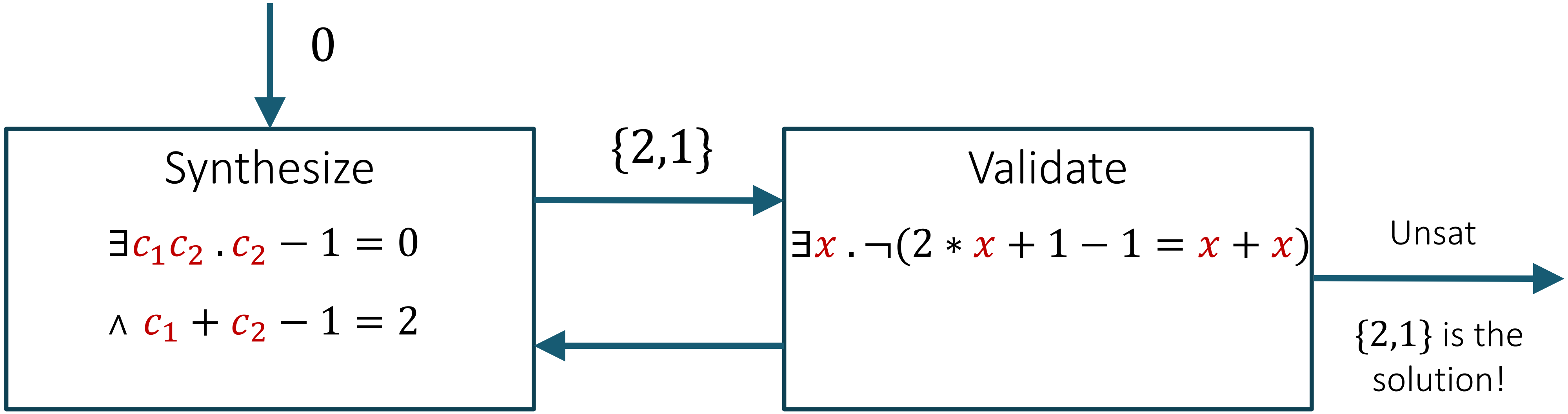
Example

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



Example

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



Program Sketching

2. How to encode the behavior of complex programs?

Symbolic execution

Behavioral constraints
= assertions / reference implementation

Structural constraints

encoding

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

3. How to solve for complex specs?

CEGIS

1. How to specify for complex programs?

Sketches

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 → sets of expressions

Expressions with `??` == sets of expressions

- linear expressions `x*?? + y*??`
- polynomials `x*x*?? + x*?? + ??`
- sets of variables `?? ? 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?

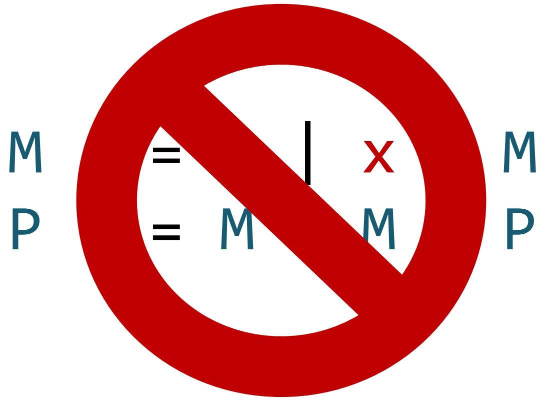
```
M ::= n | x * M
P ::= M | M + P
```

```
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

2. How to encode the behavior of complex programs?

Symbolic execution

Program c has no assertion violations on input x

Behavioral constraints = assertions / reference implementation

Structural constraints

encoding

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

3. How to solve for complex specs?

CEGIS

1. How to specify for complex programs?

Sketches

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 | e₁ + e₂
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | 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 variables to values
- $\mathcal{A}[\![\cdot]\!] : e \to \Sigma \to \mathbb{Z}$

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  :=  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?

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

Ex:

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

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
- $\mathcal{C}[\![\cdot]\!] : c \to \langle \Sigma, \Psi \rangle \to \langle \Sigma, \Psi \rangle$

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;
}
```

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

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

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

Test passed

Symbolic execution: example 1

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

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

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

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

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

Verification constraint

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

SMT solver

$\{X \mapsto 0\}$ ✗

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;
}
```

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

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

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

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

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

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

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

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

- 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;
  }
```

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

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

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

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
- $\mathcal{A}[\![\cdot]\!] : e \to \mathbf{\Phi} \to \Sigma \to \mathbb{Z}$

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

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

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

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;
}
```

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

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

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

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

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

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

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

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

Controls for generators

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

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

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

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);}
}
```

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

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

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?

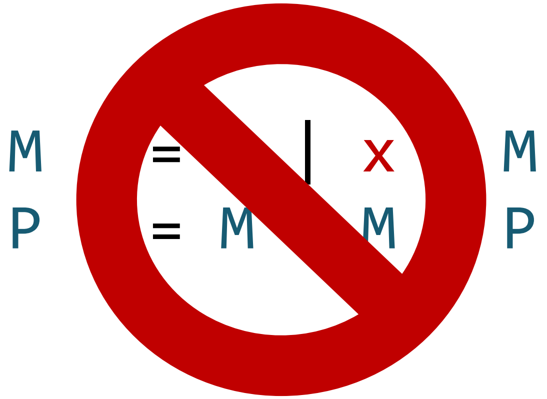
- assertions / reference implementation
- sketches
- constraint-based (CEGIS + SAT)

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

$$\mathcal{C}[\![\textbf{abort}]\!]\langle \sigma, \psi \rangle = \langle \sigma, \bot \rangle$$

CEGIS: the worst case

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

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

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

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

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

$\boxed{Q(c,x) \equiv c \neq x \vee c = 0}$ violates BOH:
no small set of counter-examples exists