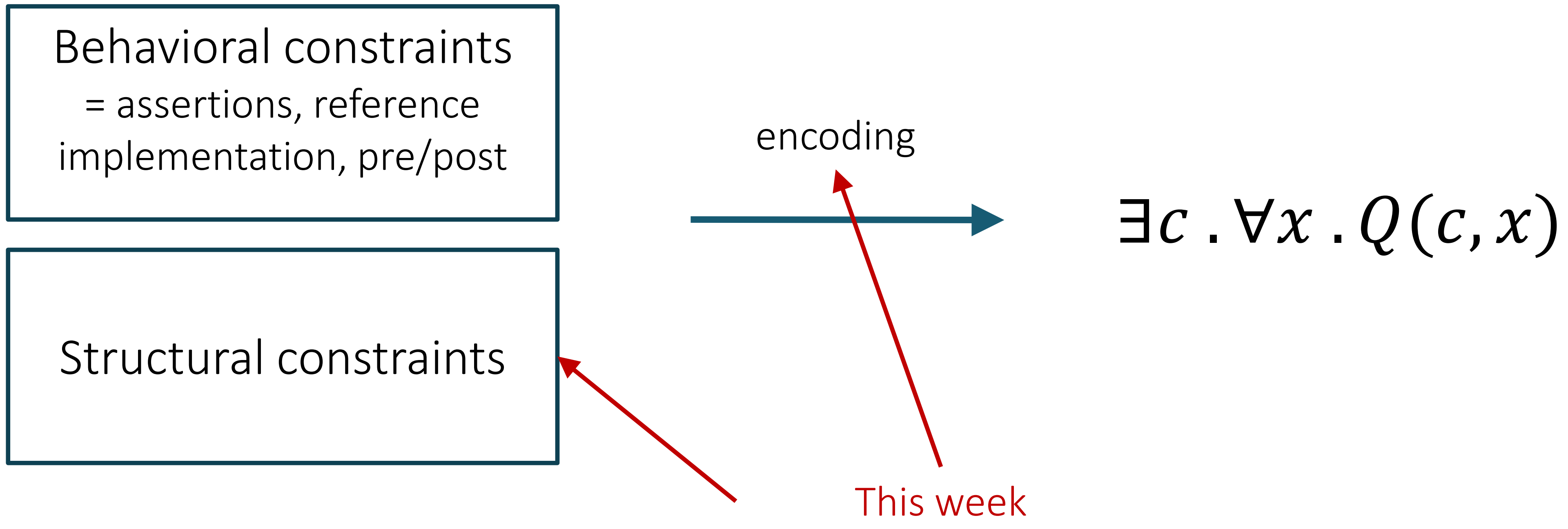
Lecture 10
Bounded Constraint-Based Synthesis

Nadia Polikarpova

Constraint-based synthesis from specifications


Behavioral constraints
= assertions, reference implementation, pre/post
Structural constraints
encoding
$\exists c\,.\,\forall x\,.\,Q(c,x)$
This week

Program sketching

Behavioral constraints
= assertions / reference implementation

Structural constraints
= sketches

$$\xrightarrow{\text{symbolic execution}} \quad \exists c \,.\, \forall x \,.\, Q(c, x)$$

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;
}
```

$\longrightarrow$

```
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 $\rightarrow$ 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 = sign()*x + sign()*y;    ➔ 1 1
    y = sign()*x + sign()*y;    ➔ 1 -1
    x = sign()*x + sign()*y;    ➔ 1 -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?

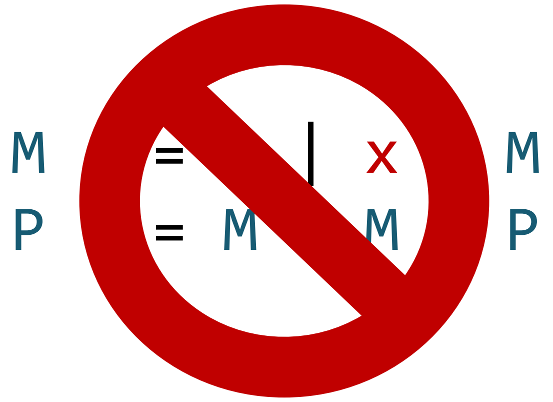
```
M ::= n | x * M
P ::= M | M + P
```

```
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);}
}
```

Encoding sketches

Behavioral constraints
= assertions / reference implementation

Structural constraints
= sketches

$$\xrightarrow{\text{symbolic execution}}$$

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

Program c has no assertion violations on input x

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:

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

  $\lambda\sigma.\lambda x.\,\mathcal{A}[\![e]\!]\sigma\ \ ?\ \ (\mathcal{C}[\![c_1]\!]\sigma)[x]\ :\ (\mathcal{C}[\![c_2]\!]\sigma)[x]$

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?

- 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:

- $\mathcal{C}[\![assert\ e]\!] = \lambda\langle \sigma, \psi \rangle . \langle \sigma, \psi \wedge \mathcal{A}[\![e]\!]\sigma = 1 \rangle$

Symbolic execution: example 1

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

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

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

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

$$\begin{tikzcd}
 & & \mathcal{C}[\![p]\!]\langle\{\}, \top\rangle = \langle\{x \to X, y \to 2X\}, 2X > X\rangle \arrow[d, "\text{Verification constraint}"] \\
\{X \mapsto 0\} & \text{✗} & \forall X.\, 2X > X \arrow[l, "\text{SMT solver}"']
\end{tikzcd}$$

Symbolic execution: example 2

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

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

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

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

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

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

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

What about loops?

Semantics of a while loop

- Let $W = \mathcal{C}[\![while\ e\ do\ c]\!]$
- W satisfies the following equation:
$$(W\ \sigma)[x] = \mathcal{A}[\![e]\!]\sigma\ \ ?\ \ (W(\mathcal{C}[\![c]\!]\sigma))[x]\ \ :\ \sigma[x]$$
- 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

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

```
→ harness 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 == 2*x;
  }
```

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

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

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

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

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

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

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

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 values are "parameterized" by the valuation of the holes
- $\mathcal{A}[\![\cdot]\!] : e \to \Sigma \to (\mathbf{\Phi} \to \mathbb{Z})$

Ex:

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

Symbolic Evaluation of Commands

Commands have two roles

- Modify the symbolic state
- Generate constraints

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

Constraints on ϕ tables, e.g.
$\lambda\phi.\phi[1] > 3$

Symbolic Evaluation of Commands

Example: assignment and assertion

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

$$\mathcal{C}[\![\textbf{assert}\ e]\!]\langle \sigma, \psi \rangle = \langle \sigma, \lambda\phi.\, \psi(\phi) \wedge \mathcal{A}[\![e]\!]\sigma\phi = 1 \rangle$$

Symbolic Evaluation of Commands

Example: conditional

$$\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!]\langle \sigma, \psi \rangle = \left\langle \begin{array}{c} \lambda x.\, \mathcal{A}[\![e]\!]\sigma \ ?\ \sigma_1[x] \ :\ \sigma_2[x], \\ \lambda \phi.\, \psi(\phi) \land \mathcal{A}[\![e]\!]\sigma \ ?\ \psi_1(\phi) : \psi_2(\phi) \end{array} \right\rangle$$

where

$$\langle \sigma_1, \psi_1 \rangle = \mathcal{C}\,[\![c_1]\!]\,\langle \sigma, \psi \rangle$$
$$\langle \sigma_2, \psi_2 \rangle = \mathcal{C}\,[\![c_2]\!]\,\langle \sigma, \psi \rangle$$

Symbolic execution of sketches: example

```
→ harness 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;
  }
```

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

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

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

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

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

$$\begin{tikzcd} & \psi = \{\neg(2 > 2) \ \wedge \ X + X = \phi_1 * X\} \arrow[d] \\ \{\phi_1 \mapsto 2\} & \exists \phi_1 . \forall X . X + X = \phi_1 * X \arrow[l, "\text{CEGIS}"'] \end{tikzcd}$$

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 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

Sketch: limitations

Everything is bounded

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

Unclear if sketching is a good user interaction model

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

CEGIS relies on the Bounded Observation Hypothesis

Sketches hard to debug

Does not prioritize likely programs

Sketch: questions

Behavioral constraints? structural constraints? search strategy?

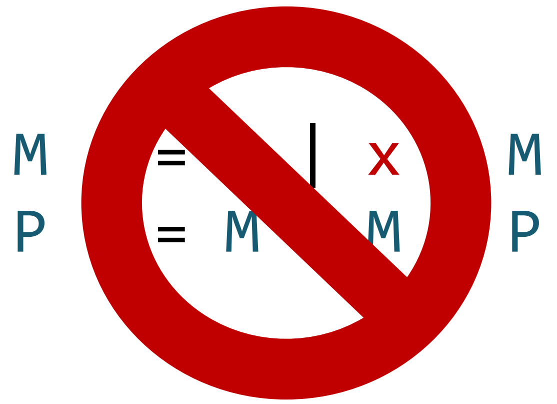
- assertions / reference implementation
- sketches
- constraint-based (CEGIS + SAT)

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

$$\mathcal{C}[\![\mathbf{abort}]\!]\langle \sigma, \psi \rangle = \langle \sigma, \lambda\phi.\,\bot \rangle$$

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

$\boxed{Q(c,x) \equiv c = 010}$ solved in single iteration

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

$\boxed{Q(c,x) \equiv x \leq c}$ does not violate BOH,
but will require 2^N iterations with worst-case counterexamples

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