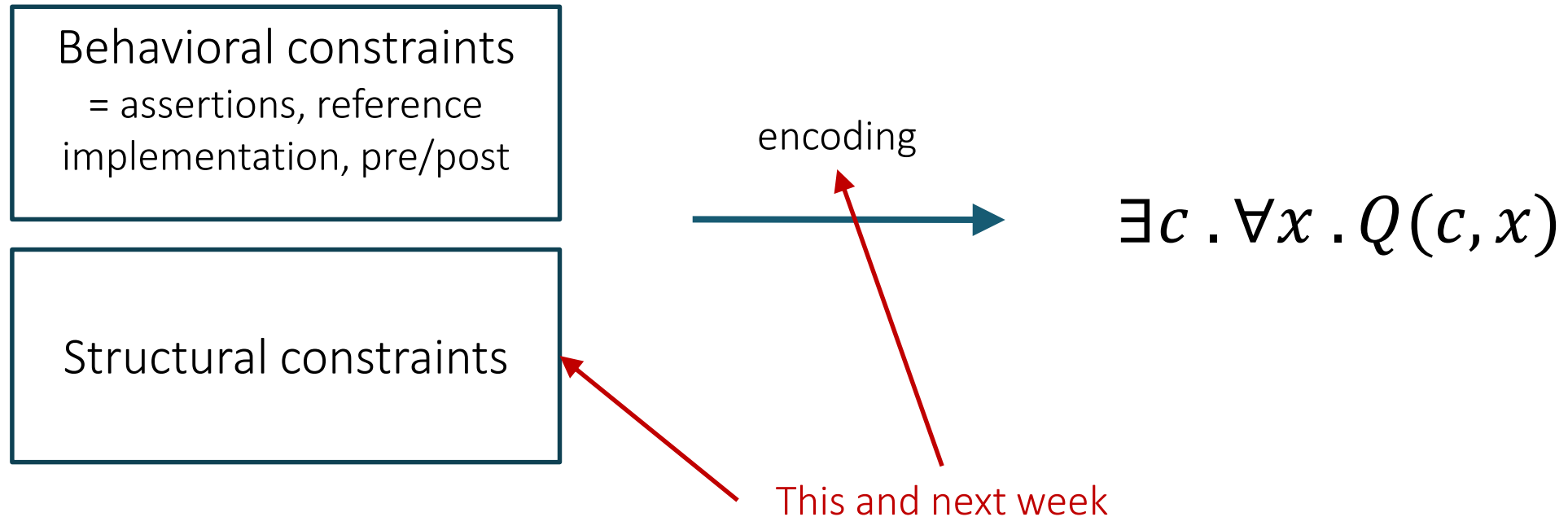


Lecture 9

Bounded Constraint-Based Synthesis

Nadia Polikarpova

Constraint-based synthesis from specifications



Program sketching

Behavioral constraints
= assertions / reference
implementation

Structural constraints
= sketches

symbolic
execution

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

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;
}
```



```
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 = 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?

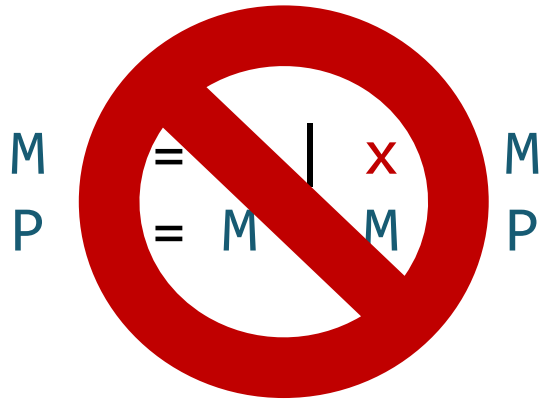
$$\begin{array}{l} M ::= n \mid x * M \\ P ::= M \mid M + P \end{array}$$

```
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);}  
}
```

Encoding sketches

Behavioral constraints
= assertions / reference
implementation

Structural constraints
= sketches

symbolic
execution

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

Semantics of a simple language

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

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
- $\mathcal{A}[\cdot] : e \rightarrow \Sigma \rightarrow \mathbb{Z}$

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

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

What does a command mean?

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

Ex:

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

What about loops?

Semantics of a while loop

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

Symbolic execution: example

```
harness void main(int x, int u){  
  int z = 0; int i = 0;  
  int y = 2 * x;  
  if (u > 0) {  
    z = 2 * x;  
  } else {  
    while (i < 2) {  
      z = z + x;  
      i = i + 1;  
    }  
  }  
  assert y == z;  
}
```

Step 1: unroll
with depth = 2

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

Symbolic execution: example

```
→ harness void main(int x, int u){  
  int z = 0; int i = 0;  
  int y = 2 * x;  
  if (u > 0) {  
    z = 2 * x;  
  } else {  
    if (i < 2) {  
      z = z + x;  
      i = i + 1;  
      if (i < 2) {  
        z = z + x;  
        i = i + 1;  
        assert !(i < 2);  
      }  
    }  
  }  
  assert y == z;  
}
```

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

$$\sigma = \{x \rightarrow X, u \rightarrow U, z \rightarrow 0, i \rightarrow 0, y \rightarrow 2X\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, z \rightarrow 2X, i \rightarrow 0, y \rightarrow 2X\}$$

$$\sigma = \{x \rightarrow X, u \rightarrow U, z \rightarrow X, i \rightarrow 1, y \rightarrow 2X\}$$

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

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

$$\sigma = \{\dots, z \rightarrow U > 0 ? 2X : X + X, i \rightarrow U > 0 ? 0 : 2, y \rightarrow 2X\}$$

$$2X = (U > 0 ? 2X : X + X)$$

Semantics of sketches

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

What does an expression mean?

- Like before, but values are “parameterized” by the valuation of the holes
- $\mathcal{A}[\cdot] : e \rightarrow \Sigma \rightarrow (\Phi \rightarrow \mathbb{Z})$

Ex:

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

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

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

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

Symbolic Evaluation of Commands

Example: conditional

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

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 u){
  int z = 0; int i = 0;
  int y = ??1 * x;
  if (u > 0) {
    z = 2 * x;
  } else {
    ...
  }
  assert y == z;
}

```

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

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

$$\sigma = \{x \rightarrow X, u \rightarrow U, z \rightarrow 2X, i \rightarrow 0, y \rightarrow \phi_1 * X\}$$

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

$$\sigma = \{\dots, z \rightarrow U > 0 ? 2X : X + X, i \rightarrow U > 0 ? 0 : 2, y \rightarrow \phi_1 * X\}$$

$$\sigma = \{\dots\}, \quad \psi = (\phi_1 * X = (U > 0 ? 2X : X + X))$$

$$\{\phi_1 \mapsto 2\} \xleftarrow{\text{CEGIS}} \exists \phi_1. \forall X U. \phi_1 * X = (U > 0 ? 2X : X + X)$$

Controls for generators

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

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

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

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

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

- what about other techniques?

Sketches hard to debug

- what about other techniques?

Sketch: questions

Behavioral constraints? structural constraints? search strategy?

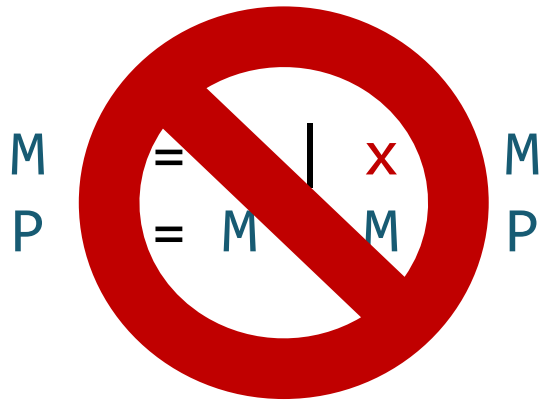
- assertions / reference implementation
- sketches
- constraint-based (CEGIS + SAT)

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)

CEGIS: the worst case

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

$$Q(c, x) \equiv (x = 000 \Rightarrow c = 111) \wedge \cdots \wedge (x = 111 \Rightarrow c = 000)$$

$$Q(c, x) \equiv x \oplus 101 = x \oplus c$$

$$Q(c, x) \equiv (c \& x \neq 000) \vee (x = 0)$$

$$Q(c, x) \equiv x \leq c$$

does not violate BOH,
but will require 2^N iterations with worst-case counterexamples

$$Q(c, x) \equiv c \neq x \vee c = 0$$

violates BOH:
no small set of counter-examples exists

Synthesis frameworks

[Torlak, Bodik '13]

Sketch

- Programming language + constant holes + generators + assertions
- Can use generator libraries to build a “synthesis DSL”

Rosette: takes it a step further

- Racket (metaprogramming!) + symbolic variables + solver queries
- Can define full-fledged SDSLs (Solver-aided DSLs)

Rosette: example

```
(require rosette/lib/synthax)
```

```
(define (poly x)  
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
```

```
(define (factored x)  
  (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??))))
```

```
(define (same p f x)  
  (assert (= (p x) (f x))))
```

not built in!
defined as a macro



```
> (define-symbolic i integer?)  
  
> (define binding  
  (synthesize #:forall (list i)  
    #:guarantee (same poly factored i)))  
  
> (print-forms binding)  
'(define (factored x) (* (+ x 0) (+ x 1) (+ x 2) (+ x 3)))
```