Lecture 9
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 and next week

Program sketching

Behavioral constraints
= assertions / reference implementation

Structural constraints
= sketches

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

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

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 → 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 abstractions 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?

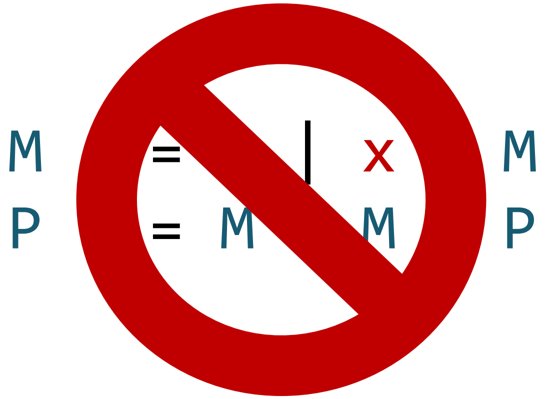
```
M ::= n | x * M
P ::= M | M + P
```

```
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);}
}
```

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)

Encoding sketches

Behavioral constraints
= assertions / reference implementation

Structural constraints
= sketches

symbolic execution →

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

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 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 | 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
- $\mathcal{C}[\![\cdot]\!] : c \to \Sigma \to \Sigma$

Ex:

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

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 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;
}
```

$U > 0$

$0 < 2$

$\neg(2 > 2)$ ✓

$2X = (U > 0\ ?\ 2X : X + X)$ ✓

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

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

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

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

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

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

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

Semantics of sketches

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 an expression mean?

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

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

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

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

Symbolic Evaluation of Commands

Example: assignment and assertion

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

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

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;
}
```

$U > 0$

$\phi_1 * X = (U > 0 \,?\, 2X : X + X)$

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

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

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

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

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

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

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

Controls for generators

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

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

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

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);}
}
```

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

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