Lecture 4
Constraint-based search

Armando Solar-Lezama and Xiaokang Qiu
Roadmap to the rest of the unit

Version Space Algebra and Lattice based Representations

Structured symbolic representations

Explicit enumeration with pruning

Constraint-based search

Stochastic search
Constraint-based search

Key idea 1:
- Search as “curve fitting”
- “curve” is a parameterized family of functions
- \[ H = \{ P[c] | c \in C \} \]

Key idea 2:
- Define a language to describe parameterized programs

Key idea 3:
- “Solve” instead of search
Synthesis with constraints

Overview of the Sketch language

Turning synthesis problems into constraints

Efficient constraint solving
Language Design Strategy

Extend base language with **one** construct

Constant hole: ??

```c
int bar (int x)
{
    int t = x * ??;
    assert t == x + x;
    return t;
}
```

Synthesizer replaces ?? with a constant

High-level constructs defined in terms of ??

```c
int bar (int x)
{
    int t = x * 2;
    assert t == x + x;
    return t;
}
```
Integer Generator → Sets of Expressions

Expressions with ?? == sets of expressions

• linear expressions: \( x^{??} + y^{??} \)
• polynomials: \( x^{x^{??}} + x^{??} + ?? \)
• sets of variables: \( ?? \ ? x : y \)
Example: Registerless Swap

Swap two words without an extra temporary

```c
int W = 32;

void swap(ref bit[W] x, ref bit[W] y) {
    if(??){ x = x ^ y; }else{ y = x ^ y; }
    if(??){ x = x ^ y; }else{ y = x ^ y; }
    if(??){ x = x ^ y; }else{ y = x ^ y; }
}

harness void main(bit[W] x, bit[W] y) {
    bit[W] tx = x; bit[W] ty = y;
    swap(x, y);
    assert x==ty && y == tx;
}
```
From simple to complex holes

We need to compose ?? to form complex holes

Borrow ideas from generative programming

• Define generators to produce families of functions
• Use partial evaluation aggressively
Generators

Look like a function
  • but are partially evaluated into their calling context

Key feature:
  • Different invocations ⇒ Different code
  • Can recursively define arbitrary families of programs
Sample Generator

```c
/**
 * Generate the set of all bit-vector expressions
 * involving +, &, xor and bitwise negation (~).
 * the bnd param limits the size of the generated expression.
 */

generator bit[W] gen(bit[W] x, int bnd) {
  assert bnd > 0;
  if(??) return x;
  if(??) return ??;
  if(??) return ~gen(x, bnd - 1);
  if(??) {
    return { | gen(x, bnd - 1) (+ | & | ^) gen(x, bnd - 1) |};
  }
}
```
Example: Least Significant Zero Bit

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??
    if(??) return ~gen(x, bnd-1);
    if(??){
        return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}

bit[W] isolate0sk (bit[W] x){
    return gen(x, 3);
}
High order generators

```
/*
 * Generate code from f n times
 * /
 generator void rep(int n, fun f){
   if(n>0){
     f();
     rep(n-1, f);
   }
 }
```
Closures + High Order Generators

generator void rep(int n, fun f) {
    if (n > 0) {
        f();
        rep(n - 1, f);
    }
}

bit[16] reverseSketch(bit[16] in) {
    bit[16] t = in;
    int s = 1;
    generator void tmp() {
        bit[16] m = ??;
        t = ((t << s) & m) | ((t >> s) & (~m));
        s = s * ??;
    }
    rep(??, tmp);
    return t;
}
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
Avoid copying and pasting
  • \texttt{repeat(n)\{ s\} \to s; s; \ldots s;}
    • each of the \( n \) copies may resolve to a distinct stmt
    • \( n \) can be a hole too.
Example: Reversing bits

```
#pragma options "--bnd-cbits 3 ";

int W = 32;

bit[W] reverseSketch(bit[W] in) {

    bit[W] t = in;
    int s = 1;
    int r = ??;
    repeat(??){
        bit[W] tmp1 = (t << s);
        bit[W] tmp2 = (t >> s);
        t = tmp1 {||} tmp2;
        // Syntactic sugar for m=??, (tmp1&m | tmp2&~m).
        s = s*r;
    }
    return t;
}
```
Framing the synthesis problem

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

```plaintext
bit[W] isolateSk (bit[W] x) implements isolate0 {
    return !(x + ??_1) & (x + ??_2) ;
}
```

• Finite set of holes so function is just a table
Framing the synthesis problem

Goal: Find a function from holes to values

- Easy in the absence of generators

```plaintext
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 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_1(x, bnd-1);
    if(??4){
        ...
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen_0(x, 3);
}
Framing the synthesis problem

Generators need something more

```c
generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(\phi(??_1)) return x;
    if(\phi(??_2)) return \phi(??_3);
    if(\phi(??_3)) return \sim gen_1(x, bnd-1);
    if(\phi(??_4)){
        ...
    }
}

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

Generators need something more
  • The value of the holes depends on the context
Framing the synthesis problem

- Potentially unbounded set of unknowns
- We can bound the depth of recursion
  - That means again $\phi$ is just a table

```c
generator bit[W] gen(context $\tau$, bit[W] x, int bnd) {
    assert bnd > 0;
    if($\phi(\tau,??_1)$) return x;
    if($\phi(\tau,??_2)$) return $\phi(\tau,??_5)$;
    if($\phi(\tau,??_3)$) return $\neg gen_{g_1}(\tau \cdot g_1, x, bnd-1$);
    if($\phi(\tau,??_4)$) {
        return $\{ \mid gen_{g_2}(\tau \cdot g_2, x, bnd-1) (+ \mid & \mid ^) \mid gen_{g_3}(\tau \cdot g_3, x, bnd-1) \mid \}$;
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen_{g_0}(g_0, x, 3);
}

$\phi(g_0,??_k)$
$\phi(g_0g_1,??_k)$
$\phi(g_0g_2,??_k)$
$\phi(g_0g_1g_2,??_k)$
$\phi(g_0g_1g_3,??_k)$
$\phi(g_0g_1g_2g_1,??_k)$
...
```
The inductive synthesis problem

\[ \exists c \ \forall in \in E \ Q(in, c) \]

where \( E = \{x_1, x_2, \ldots, x_k\} \)