SYNTHESIS WITH PARTIAL PROGRAMS
The synthesis problem

Synthesis as a game

For every move of the environment

Synthesized program makes a counter move
The challenge of synthesis

For functions, the environment sets the inputs
- i.e. whatever we synthesize must work for all inputs

Modeled with a doubly quantified constraint

\[ \exists P \forall \text{in} \ (\text{in}, P \models \text{Spec}) \]
- What does it mean to quantify over programs?
Quantifying over programs

Synthesis as curve fitting

• we want a function that satisfies some properties

It’s hard to do curve fitting with arbitrary curves

• Instead, people use parameterized families of curves

• Quantify over parameters instead of over functions

\[ \exists c \forall in \ (in, P[c] \models Spec) \]

Key idea:
Let user define parameterized functions with partial programs
THE SKETCH SYNTHESIS INFRASTRUCTURE
Synthesizer Structure
DESIGNING A LANGUAGE FOR PARTIAL PROGRAMS
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 $\rightarrow$ Sets of Expressions

Expressions with ?? $==$ sets of expressions

- linear expressions $\quad x^{*??} + y^{*??}$
- polynomials $\quad x^{*x*??} + x^{*??} + ??$
- sets of variables $\quad ?? \ ? x : y$
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;
}
```
Syntactic Sugar

```plaintext
{ | RegExp | }
```

RegExp supports choice ‘|’ and optional ‘?’

- can be used arbitrarily within an expression
  - to select operands
    ```plaintext
    { | (x | y | z) + 1 | }
    ```
  - to select operators
    ```plaintext
    { | x (+ | -) y | }
    ```
  - to select fields
    ```plaintext
    { | n(.prev | .next)? | }
    ```
  - to select arguments
    ```plaintext
    { | 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

- `repeat(n){ s} \Rightarrow s; s; \ldots s;`
  
- each of the $n$ copies may resolve to a distinct stmt
- $n$ can be a hole too.
Example: Reversing bits

```c
bit[W] reverse (bit[W] in) {
    bit [W] out=0;
    for (int i = 0; i < W; i++) {
        out[i] = in[W-1 - i];
    }
    return out;
}
```
Example: Reversing bits

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

int W = 32;

bit[W] reverseSketch(bit[W] in) implements reverse {
  bit[W] t = in;
  int s = 1;
  int r = ??;
  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;
}
```
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 $\rightarrow$ Different code
  • Can recursively define arbitrary families of programs
Example: Least Significant Zero Bit

- 0010 0101 \rightarrow 0000 0010

```
int W = 32;

bit[W] isolate0 (bit[W] x) { // W: word size
    bit[W] ret = 0;
    for (int i = 0; i < W; i++)
        if (!x[i]) { ret[i] = 1; return ret; }
}
```

Trick:
- Adding 1 to a string of ones turns the next zero to a 1
- i.e. 000111 + 1 = 001000
/**
 * 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

```c
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) implements isolate0 { 
    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) implements reverse {
    bit[16] t = in;
    int s = 1;
    generator void tmp() {
        t = (t << s) { | } (t >> s);
        s = s*??;
    }
    rep(??, tmp);
    return t;
}