Roadmap

Previously
- Inductive synthesis
- CFGs (SyGuS) and DSLs (FlashFill)
- Enumerative search and Representation-based search

Today
- Inductive synthesis → Functional synthesis
- Generators (Sketch)
- Constraint-based Search
User intent

Input-output examples

Search space

- Grammars
- DSLs

Search strategy

- Enumerative search
- Representation-based (version space algebras)
Constraint-based search: Sketch

Key idea 1:
- Search as “curve fitting”
- “curve” is a parametric family of programs

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

Key idea 3:
- “Solve” instead of search
Sketch: a language for parametric programs

Turning synthesis problems into constraints

Efficient constraint solving
Sketch: a language for parametric programs

Turning synthesis problems into constraints

Efficient constraint solving
Language design strategy

Extend base language with *one* construct
Unknown constant hole: ??

Type inferred from context

Synthesizer replaces ?? with an integer constant

High-level constructs defined in terms of ??
harness void test() {
    bar(2);
    bar(10);
    bar(1);
}

int bar (int x) {
    int t = x * 2;
    assert t == x + x;
    return t;
}

harness void test() {
    bar(2);
    bar(10);
    bar(1);
}

int bar (int x) {
    int t = x * 2;
    assert t == x + x;
    return t;
}

Assertion validity is modulo test harness
Inductive synthesis: no inputs to harness
Integer holes $\rightarrow$ sets of expressions

Expressions with ?? = sets of expressions

- linear expressions $x^{\text{??}} + y^{\text{??}}$
- polynomials $x^{x^{\text{??}}} + x^{\text{??}} + ??$
- sets of variables $?? \ ? x : y$
Example: Least significant zero bit

0010 0101 → 0000 0010
0010 0000 → 0000 0001

```c
harness void test()
{
    assert isolateSk(00100101) = 00000010;
    assert isolateSk(00100000) = 00000000;
}
```
Example: Least significant zero bit

0010 0101 → 0000 0010

0010 0000 → 0000 0001

Trick for an optimized program:
Adding 1 to a string of ones turns the next zero to a one:
000111 + 1 = 001000

```c
harness void test()
{
    assert isolateSk(00100101) = 00000010;
    assert isolateSk(00100000) = 00000000;
}

int isolateSk(bit[W] x) {
    int t = !(x + ??) & (x + ??);
    return t;
}
```
Example: Least significant zero bit

```
0010 0101 \rightarrow 0000 0010
0010 0000 \rightarrow 0000 0001
```

*Trick for an optimized program:*

Adding 1 to a string of ones turns the next zero to a one:

```
000111 + 1 = 001000
```

```c
harness void test()
{
    assert isolateSk(00100101) = 00000010;
    assert isolateSk(00100000) = 00000000;
}

bit[W] isolateSk (bit[W] x) {
    int t = !(x + ??) & (x + ??); 
    return t;
}
```
Integer holes $\rightarrow$ sets of expressions

Expressions with ?? = sets of expressions

- linear expressions
  - $x*?? + y*??$
- polynomials
  - $x*x*?? + x*?? + ??$
- sets of variables
  - $?? ? x : y$

Semantically powerful but syntactically clunky

Regular Expressions are a more convenient way of defining sets
Regular expression generators (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
Least significant zero bit revisited

How did I know the solution would take the form !(x + ??) & (x + ??)

What if all you know is that the solution involves x, +, & and !

```
bit[W] tmp=0;
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
return tmp;
```

This is now a set of statements (and a really big one too)
Sets of statements

Statements with holes = sets of statements

Higher level constructs for statements too: repeat

```c
bit[W] tmp=0;
repeat(3){
    { | x | tmp |} = { | (!)?(x | tmp) (& | +) (x | tmp | ??) |};
}
return tmp;
```
Avoid copying and pasting

- \( \text{repeat}(n) \{ s \} \Rightarrow s; s; \ldots s; \) 
  - each of the \( n \) copies may resolve to a distinct stmt
  - \( n \) can be a hole too.

\begin{verbatim}
bit[W] tmp=0;
repeat(??){
    { | x | tmp | } = { | (!)?((x | tmp) (& | +) (x | tmp | ??)) | };
}
return tmp;
\end{verbatim}

The synthesizer won’t try to minimize \( n \)
Use \(--unrollamnt\) to set the maximum value of \( n \)
Procedures and Sets of Procedures

Two types of procedures
  ▪ standard procedures
    ▪ represents a single procedure
    ▪ all call sites resolve to the same procedure
    ▪ identified by the keyword static
  ▪ generators
    ▪ represents a set of procedures
    ▪ each call site resolves to a different procedure in the set
    ▪ can recursively define arbitrary families of programs
    ▪ default in the Sketch implementation

Generators are very expressive!
Example: Least significant zero bit

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

bit[W] tmp=0;
repeat(??){
    { | x | tmp | } = { | (!)?((x | tmp) (& | +) (x | tmp | ??)) | };
}
return tmp;
High order generators

```c
/*
 * Generate code from f n times
 */
generator void rep(int n, fun f){
    if(n>0){
        f();
        repeat(n-1, f);
    }
}
```
Defining sets of code fragments is the key to Sketching effectively.
- Overview of the Sketch language
- Turning synthesis problems into constraints
- Efficient constraint solving
Step 1: Turn holes into special inputs

The ?? operator is modeled as a special control input

```c
#define W

bit[W] isolSk(bit[W] x) {
    return ~(x + ??) & (x + ??);
}
```

Bounded candidate spaces are important
- bounded unrolling of `repeat` is important
- bounded inlining of generators is important
Step 2: Constraining the set of controls

- Correct control
  - causes the spec & sketch to match for all inputs
  - causes all assertions to be satisfied for all inputs

- Constraints are collected into a predicate
  \[ Q(x, c) \]

- `-showDAG` will show you the constraints!
A Sketch as a constraint system

Synthesis reduces to constraint satisfaction

∃c. ∀x in test harness. Q(x, c)

Constraints are too hard for standard techniques
- Universal quantification over inputs
- Too many inputs
- Too many constraints
- Too many holes
Overview of the Sketch language

Turning synthesis problems into constraints

Efficient constraint solving
User intent
- Input-output examples
- Logical specifications
- Equivalent programs

Search space
- Generators

Search strategy
- Constraint-based search

See also "rements"
A Sketch as a constraint system

Synthesis reduces to constraint satisfaction

\[ \exists c \forall x. \ Q(x, c) \]

Constraints are too hard for standard techniques
- Universal quantification over inputs
- Too many inputs
- Too many constraints
- Too many holes

Technique 1: Quantifier Elimination
- Can blow up

Technique 2: CEGIS
Insight

Sketches are not arbitrary constraint systems
  ▸ They express the high level structure of a program

A small set of inputs can fully constrain the solution
  ▸ focus on corner cases

\[ \exists c. \forall x \text{ in } E. \quad Q(x, c) \]

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

This is an inductive synthesis problem!
  ▸ how do we find the set \( E \)?
  ▸ how do we solve the inductive synthesis problem?
Step 3: Counterexample Guided Inductive Synthesis

Idea: Couple inductive synthesizer with a verifier
- Verifier is charged with detecting convergence

- Inductive Synthesizer
  - Derive candidate implementation from concrete inputs.
    \[ \exists c. \forall x \in E. \ Q(x, c) \]

- Verifier
  - Your verifier goes here

- Observation set \( E \)

- Candidate implementation

- Standard implementation uses Sat based bounded verifier
- Any verifier/checker that produces counterexamples works
Summary

Today
- Inductive synthesis → Functional synthesis
- Generators (Sketch)
- Constraint-based Search

Next
- Another constraint-based approach for functional synthesis