Introduction to Synthesis

Alvin Cheung
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Based on slides from Prof. Solar-Lezama

December 9, 2013
What do we mean by synthesis

The promise:

Automate the task of writing programs
What do we mean by synthesis

We want to get code from high-level specs
  - Python and VB are pretty high level, why is that not synthesis?

Support compositional and incremental specs
  - Python and VB don’t have this property
    • If I don’t like the way the python compiler/runtime is implementing my program, I am out of luck.
  - Logical specifications do
    • I can always add additional properties that my system can satisfy
  - Specs are not only functional
    • Structural specifications play a big role in synthesis
    • How is my algorithm going to look like.
The fundamental challenge of synthesis is dealing with an uncooperative environment

- For reactive systems, people model this as a game
  - For every move of the adversary (ever action of the environment), the synthesized program must make a counter-move that keeps the system working correctly.
  - The game can be modeled with an automata
The fundamental challenge of synthesis is dealing with an uncooperative environment

- If we are synthesizing functions, the environment provides the inputs
  - i.e. whatever we synthesize must work correctly for all inputs

- This is modeled with a doubly quantified constraint
  - E.g. if the spec is given as pre and post conditions, we have

\[
\exists P \forall \sigma \ (\sigma \models \{pre\}) \Rightarrow (\sigma \models WP(P,\{post\}))
\]

- What does it mean to quantify over the space of programs?
Quantifying over programs

Synthesis in the functional setting can be seen as curve fitting
- i.e. we want to find a curve that satisfies some properties

It’s very hard to do curve fitting when you have to consider arbitrary curves
- Instead, people use *parameterized* families of curves
- This means you quantify over parameters instead of over functions

This is the first fundamental idea in software synthesis
- People call these Sketches, scaffolds, templates, ...
- They are all the same thing
Solving the synthesis problem

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

Many ways to represent \((in, P[c] \models Spec)\)
- Important area of research
- At the abstract level, it’s just a predicate
Many different options

a) Eliminate $\forall$ symbolically
   - You can use Farkas Lemma
   - You can use an abstract domain
   - You can use plain-vanilla elimination (not recommended)

b) Sample the space of inputs intelligently
Hypothesis

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

A small number of inputs can be enough
  ▪ focus on corner cases

\[ \exists c \forall in \in E \ Q(in, c) \]
  where \( E = \{x_1, x_2, \ldots, x_k\} \)

This is an inductive synthesis problem!
The Sketch Language

Define parameterized programs explicitly

- Think of the parameterized programs as “programs with holes”

Example: Hello World of Sketching

```
spec:
int foo (int x)
{
  return x + x;
}

sketch:
int bar (int x) implements foo
{
  return x * ??;
}
```

Integer Hole
Language support for defining \( \{P_c\} \)
- Simple integer hole (??) + *generators*

Constraints generated incrementally by *CEGIS*

**Ex: Sketch**

\[ \exists c \ s.t. \ Correct(P_c, \text{in}_i) \]

\[ \exists \text{in} \ s.t. \ \neg Correct(P_c, \text{in}_i) \]

Insert your favorite checker here
CEGIS in Detail

\[ Q(c, in) \]

**Synthesize**

\[
Q(c, in_0) \quad Q(c, in_1) \\
Q(c, in_2) \quad Q(c, in_3)
\]

\[ \{in_i\} \]

**Check**

\[
\neg Q(c, in_1) \quad \neg Q(c, in_3) \\
\neg Q(c, in_2) \quad \neg Q(c, in_4)
\]
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

```c
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
Sample Generator

/**
 * 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) implements isolate0 {
    return gen(x, 3);
}
Framing the synthesis problem

Goal: Find a function from holes to values

- Easy in the absence of generators

```c
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 + \phi(??_1)) \& (x + \phi(??_2)) ;
}
```

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

Generators need something more

```c
generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??1) return x;
    if(??2) return ??5;
    if(??3) return ~gen_{g1}(x, bnd-1);
    if(??4){
        ...
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen_{g0}(x, 3);
}
```
Framing the synthesis problem

Generators need something more

generator bit[W] gen(bit[W] x, int bnd){
  assert bnd > 0;
  if(\phi(??_1)) return x;
  if(\phi(??_2)) return \phi(??_5);
  if(\phi(??_3)) return \neg gen_g1(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

```c
generator bit[W] gen(context τ, bit[W] x, int bnd) {
    assert bnd > 0;
    if(φ(τ,??_1)) return x;
    if(φ(τ,??_2)) return φ(τ,??_5);
    if(φ(τ,??_3)) return ~geng1(τ \cdot g_1, x, bnd-1);
    if(φ(τ,??_4)){
        ...
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen_g0(g_0, x, 3);
}
```
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($\phi(\tau, {???})$) return x;
    if($\phi(\tau, {???})$) return $\phi(\tau, {???})$;
    if($\phi(\tau, {???})$) return $\neg gen_{g_1}(\tau \cdot g_1, x, bnd-1)$;
    if($\phi(\tau, {???})$) {
        return { $\mid gen_{g_2}(\tau \cdot g_2, x, bnd-1) (+ \mid & \mid ^) gen_{g_3}(\tau \cdot g_3, x, bnd-1) |$ };
    }
}

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

$\phi(g_0, {???})$
$\phi(g_0g_1, {???})$
$\phi(g_0g_2, {???})$
$\phi(g_0g_1g_2, {???})$
$\phi(g_0g_1g_3, {???})$
$\phi(g_0g_1g_2g_1, {???})$
...