Beyond Verification

Software Synthesis
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

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

- 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
The Sketch Language

Define parameterized programs explicitly

- Think of the parameterized programs as "programs with holes"

Example: Hello World of Sketching

\[
\text{spec:}
\begin{align*}
\text{int foo (int x) } & \\
\{ & \\
\quad \text{return x + x; } & \\
\}
\end{align*}
\]

\[
\text{sketch:}
\begin{align*}
\text{int bar (int x) implements foo } & \\
\{ & \\
\quad \text{return x * ??; } & \\
\}
\end{align*}
\]

Integer Hole
Expressions with ?? == sets of expressions
- linear expressions  \( x^*?? + y^*?? \)
- polynomials  \( x*x^*?? + x*?? + ?? \)
- sets of variables  ?? ? x : y
**Example: Least Significant Zero Bit**

- 0010 0101 → 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

$$!(x + ??) \& (x + ??) \rightarrow !(x + 1) \& (x + 0) \quad !(x + 1) \& (x + \text{0xFFFF})$$

$$!(x + 0) \& (x + 1) \quad !(x + \text{0xFFFF}) \& (x + 1)$$
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; }
}

bit[W] isolateSk (bit[W] x) implements isolate0 {
    return !(x + ??) & (x + ??);
}
```
Integer Holes → 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

\{ | \text{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 One 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){
    repeat(3){
        {% x | tmp |} = {% (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
    }
}
return tmp;
```
Avoid copying and pasting

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

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

Keep in mind:
- the synthesizer won’t try to minimize n
Solving for a parameterized program

At a high level, two fundamental approaches:

- Search and Test

- Derive in one shot
  - Usually by means of abstraction.
The CEGIS approach

Synthesis reduces to constraint satisfaction

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

Constraints are too hard for standard techniques

- Universal quantification over inputs
- Too many inputs
- Too many constraints
- Too many holes
Insight

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

\[ \exists \phi. \ \forall x \in E. Q(x, \phi) \]
where \( E = \{x_1, x_2, ..., x_k\} \)

This is an inductive synthesis problem!
CEGIS Synthesis algorithm

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

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

Synthesize

Check
Insert your favorite checker here

\{in_i\} \rightarrow c \rightarrow \exists in \text{ s.t. } \neg \text{Correct}(P_c, in_i) \rightarrow in \rightarrow \{in_i\}
$Q(c, in)$

Synthesize

$Q(c, in_0)$  $Q(c, in_1)$

$Q(c, in_2)$  $Q(c, in_3)$

Check

$\neg Q(c, in_2)$

A sketch as a constraint system

```c
int lin(int x){
    if(x > ??1)
        return ??2*x + ??3;
    else
        return ??4*x;
}

void main(int x){
    int t1 = lin(x);
    int t2 = lin(x+1);
    if(x<4) assert t1 >= x*x;
    if(x>=3) assert t2-t1 == 1;
}
```
Ex: Population count. 0010 0110 → 3

```c
int pop (bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        n(x[i]) count++;
    }
    return count;
}
```

\[ F(x) = \]
int popSketched (bit[W] x)
    implements pop {
    repeat(??) {
        x = (x & ??)
        + ((x >> ??) & ??);
    }
    return x;
}
6.820 Fundamentals of Program Analysis
Fall 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.