CSC410
Program Synthesis

AZADEH FARZAN
FALL 2017
The idea

Produce small code fragments that satisfy the given specification

As a programming aid:
- It helps you write programs.
- The insight is yours, the tedious work is done by the tool.
Looks familiar?

Produce small code fragments that satisfy the given specification

Does this look like something you know in the context of programming?
Compilation vs Synthesis

**Compilation:** it does not produce unknowns just translates.

- Represent **source** program as abstract syntax tree (AST)
- **Lower** the AST from source to target language
- Lowering performed with tree **rewrite rules**, which are guaranteed to be correct.

**Synthesis:**

- Similar mechanism: start from **spec**, and use rules to **lower** the spec to a desired program.
- Rewrite sequence can be **non-deterministic**
- Rewrite rules **need not** be arbitrarily **composable**.
Let’s start with a concrete example ...
Sketching
What is the interface like?
Setup

Sketches:
- Programs with holes.
- Write what you know, use the holes for the details you do not want to fully figure out.

Fundamentals:
- How does the programmer communicate what he wants?
- How does the tool define a universe of “things” that can fill up a hole?
Program Synthesis Problem

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

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

**result:**
```plaintext
int bar (int x) implements foo {
    return x << 1;
}
```
Stepping back to a general setting ...
What is program synthesis?

Say we have a property $\phi$ over input/output pairs for a program.

We can formally frame the problem of finding a function that fits this specification through:

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

When does automated programming make sense?

- when it is easier to write $\phi$ than it is to write $P$.
- when it is easier to be convinced about $\phi$ than it is to write $P$. 
What is program synthesis?

Say we have a property \( \phi \) over input/output pairs for a program.

We can formally frame the problem of finding a function that fits this specification through:

\[
\exists P. \forall x. \phi(x, P(x))
\]

Generally, this is complex constraint solving problem, where generic efficient solutions do not exist.

**Algorithmic challenge:** solve this quantified satisfiability problem by reducing it to a flat satisfiability problem for which efficient solvers exist.
Going back into the special setting of Sketching ...
Program Synthesis Problem

**spec:**

```java
int foo (int x) {
    return x + x;
}
```

**sketch:**

```java
int bar (int x) implements foo {
    return x << ??;
}
```

**result:**

```java
int bar (int x) implements foo {
    return x << 1;
}
```
We can use *equivalence to existing code* as specification.

We can also use *logical specification*, exactly as in verification.

We can also use *samples of input/out pairs* as specification.
Sketch

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

**Holes** are meant to be placeholder for code fragments.

**Fragments** range over a user-defined set.

---

Defining a good set of fragments is the key to effective Sketching!
Fragments range over a user-defined set.

Sets are defined hierarchically.
Example

Least significant non-zero bit: 0010 1010 → 0000 0010

Trivial Code:

```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;  }
}
```

Insight: adding 1 to a string of 1’s turns the next 0 to 1.
Example

Least significant zero bit: 0010 0101 \rightarrow 0000 0010

Trivial code:

```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; }
}

sketch of better code:

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

!(x + 0) & (x + 1)
Too much to know?
Example

Assume all we know is that the solution uses x, +, &, and !.

**Looser Sketch:**

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

Regular expressions are used to define expressions and choice in code fragments.

**Even more expressive:**

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

The language can be made very rich. What is stopping us?
Algorithmic Program Synthesis
A partial program defines a search space

Goal: find a candidate in the space that satisfies $\phi$

The space is huge!

Try to describe it symbolically and use the power of solvers to do the search.
Search Over Candidate Programs

- **spec**
- **sketch**

**program-to-formula translator**

**solver**

"synthesis engine"

**code generator**

- sketch $P[h]$
- $h \mapsto 1$

$\phi$

$P[1]$
How does this work?
Program as a Formula

We have a program:

\[ f(x) \{ \text{return } x + x \} \]

And a formula that represents it:

\[ S_f(x, y) : y = x + x \]

Now a solver is an interpreter:

\[ S_f(x, y) \land x = 3 \]

\[ y \mapsto 6 \]

And, a program inverter:

\[ S_f(x, y) \land y = 6 \]

\[ x \mapsto 3 \]

This bidirectionally enables synthesis!
Constraints Solving for Search

We have a spec and a partial program:

```plaintext
spec(x) { return x + x}
sketch(x) { return x << ?? }
```

The solver finds $h$, and therefore, synthesizes the program:

$$S_{\text{sketch}}(x, y, h) : y = x \times 2^h$$

We may not always get lucky with the choice of input:

$$S_{\text{sketch}}(x, y, h) \land x = 2 \land y = 4 \quad h \mapsto 1$$

$$S_{\text{sketch}}(x, y, h) \land x = 0 \land y = 0 \quad h \mapsto 1, 2, 3, 4, \ldots$$

$$\land S_{\text{sketch}}(x', y', h) \land x' = 3 \land y' = 6 \quad h \mapsto 1$$
Inductive Synthesis

Small world hypothesis:

There is a small set of inputs where if the program is correct for these, then it is correct for every input.

So, instead of solving this:

$$\exists h \forall x. \phi(x, P(x, h))$$

We solve this:

$$\exists h. \phi(x_1, P(x_1, h)) \land \cdots \land \phi(x_n, P(x_n, h))$$

But where do these magical inputs come from?
CounterExample-Guided Inductive Synthesis (CEGIS)

**Inductive Synthesizer**
compute a candidate implementation from concrete inputs.

\((x_1, o_1), \ldots, (x_k, o_k)\)

**verifier/checker**
succeed

candidate implementation

fail

add a (bounded) counterexample input
Small World Hypothesis

\[ C = \text{size of candidate space} = \exp(\text{bits of controls}) \]
Other Types of Program Synthesis

- Counterexample-guided Inductive Synthesis
- Controller synthesis
- Deductive Synthesis
Controller Synthesis

**Given a plant** $P$ **and a specification** $\phi$, **is there a controller** $C$ **such that the system** $C \parallel P$ **satisfies** $\phi$? 

---

**Plant**: 2-players game arena

**Specification**: game objective for Player 1

**Controller**: winning strategy for Player 1
We may come back to this when we study model checking
Deductive Synthesis

\[ \forall n \cdot 2^n = 2^{**n} \]

\[ \forall k, n \cdot k \cdot 2^n = k << n \]

\[ \forall k, n :: k \cdot 4 + n = \text{s4addl}(k, n) \]

\[
\begin{align*}
\text{reg6} & \rightarrow \text{s4addl} \\
\text{reg6} \cdot 4 + 1 \quad \text{specification} & \rightarrow \text{s4addl(\text{reg6},1)} \quad \text{synthesized program}
\end{align*}
\]
Deductive Synthesis

There are two types of axioms:

- Instruction Semantics
- Algebraic Properties: associativity, commutativity, ....

This works very well, as long as rules can be defined so that what we are looking for exists within the space defined.
This is in the spirit of compilation but far more sophisticated ...
A Non-trivial Example
Example: Maximum Contiguous Sum in Linear Time

```
#pragma options "--bnd-inline-amnt 5 --bnd-inbits 3 --bnd-cbits 2"

int W = 4;

int lss(int[W] in) {
    for(int i=0; i<W; ++i) { in[i] = in[i] - 3; }
    int sum = 0;
    int maxsum = 0;
    for(int i = 0;i<W; i++) {
        sum = 0;
        for(int j = i; j<W; j++) {
            sum = sum + in[j];
            if(sum > maxsum) maxsum = sum;
        }
    }
    return maxsum;
}
```
Example: Maximum Contiguous Sum in Linear Time

```c
int lssSketch(int[W] in) implements lss {
    for(int i=0; i<W; ++i){ in[i] = in[i] - 3; }
    int sum = 0;
    int maxsum = 0;
    int psum = 0;
    for(int i = 0; i<W; i++) {
        psum = sum;
        sum = sum + in[i];
        if({ (sum | psum | ?) (+|-) (sum | psum | ?) ≤ ?? |})
            sum = { (sum | psum | ?) (+|-) (sum | psum | ?) |};
        if(sum > maxsum) maxsum = sum;
    }
    return maxsum;
}
```

Did you manage to produce the answer on your own? NO? then Google it !!!
SKETCH