Lecture 8
Functional synthesis

Armando Solar-Lezama and Xiaokang Qiu
Moving beyond Inductive synthesis

We want to synthesize functions that satisfy richer correctness criteria

Key questions

• Specification formalisms
  • How do we describe the intended behavior?

• How do we ensure correctness
  • No longer trivial as in the inductive case

• Synthesis approaches
  • Deductive vs. inductive
The general synthesis problem

\[ \exists P \ \forall \text{in} \ (\text{in}, P \models \text{Spec}) \]
The general synthesis problem

\[ \exists c \forall in \ (in, Sk(c) \models Spec) \]
The general synthesis problem
Defining correctness

out=Sort(n, in)
Defining correctness

\[ n \geq 0 \land n \leq \text{len}(\text{in}) \]

\[ \text{out} = \text{Sort}(n, \text{in}) \]
Defining correctness

Precondition: \( n \geq 0 \land n \leq \text{len}(\text{in}) \)

\( \text{out} = \text{Sort}(n, \text{in}) \)
Defining correctness

Precondition: \[ n \geq 0 \land n \leq \text{len}(\text{in}) \]

Postcondition: \[ \text{out} = \text{Sort}(n, \text{in}) \]
Defining correctness

Precondition: \[ n \geq 0 \land n \leq \text{len}(\text{in}) \]
\[ \text{out} = \text{Sort}(n, \text{in}) \]

Postcondition: \[ \forall i. 0 \leq i < n - 1 \Rightarrow \text{out}[i] \leq \text{out}[i + 1] \]
Defining correctness

Precondition: \[ n \geq 0 \land n \leq \text{len}(\text{in}) \]
\[ \text{out} = \text{Sort}(n, \text{in}) \]

Postcondition: \[ \forall i. \ 0 \leq i < n - 1 \Rightarrow \text{out}[i] \leq \text{out}[i + 1] \]
\[ \text{len(out)} = n \]
Defining correctness

Precondition: \( n \geq 0 \land n \leq \text{len}(\text{in}) \)

\( \text{out} = \text{Sort}(n, \text{in}) \)

Postcondition: \( \forall i. 0 \leq i < n - 1 \Rightarrow \text{out}[i] \leq \text{out}[i + 1] \)

\( \text{len}(\text{out}) = n \)

\( \forall i. 0 \leq i < n - 1 \Rightarrow \exists j \in [j] = \text{out}[i] \)
Defining correctness

Precondition: \[ n \geq 0 \land n \leq \text{len}(in) \]
\[ \text{out} = \text{Sort}(n, \text{in}) \]

Postcondition:
\[ \forall i. \, 0 \leq i < n - 1 \Rightarrow \text{out}[i] \leq \text{out}[i + 1] \]
\[ \text{len(out)} = n \]
\[ \forall i. \, 0 \leq i < n - 1 \Rightarrow \exists j \, \text{in}[j] = \text{out}[i] \]
\[ \forall i. \, 0 \leq i < n - 1 \Rightarrow \exists j \, \text{in}[i] = \text{out}[j] \]
Defining correctness

Precondition: \( n \geq 0 \land n \leq \text{len}(\text{in}) \)

\[ \text{out} = \text{Sort}(n, \text{in}) \]

Postcondition: \( \forall i. 0 \leq i < n - 1 \Rightarrow \text{out}[i] \leq \text{out}[i + 1] \)

\( \text{len}(\text{out}) = n \)

\( \forall i. 0 \leq i < n - 1 \Rightarrow \exists j. \text{in}[j] = \text{out}[i] \)

\( \forall i. 0 \leq i < n - 1 \Rightarrow \exists j. \text{in}[i] = \text{out}[j] \)

\( \exists P : \text{int} \rightarrow \text{int}. \forall i. i < n \Rightarrow \text{out}[i] = \text{in}[P(i)] \land \exists j. P(j) = i \)
Multimodal Synthesis

Trick:
combine many simple specs in different formalisms to fully constrain the behavior

- Concrete scenarios
- Abstract scenarios
- Partial specs
- Safety properties
- Structural Info

Synthesizer

Correct Code
Ensuring correctness

This is a hard problem in general

\[ \forall \, \text{in} \quad (\text{in}, P \models \text{Spec}) \]

Two points of view:

- Not my problem
  - That’s what 6.820 is for

- Synthesis can make verification simpler
  - Synthesize code that is easier to prove correct
  - Ensure correctness by construction
Counterexample guided inductive synthesis

Ideas
- Rely on an oracle to tell you if your program is correct
- If it is not, rely on oracle to generate counterexample inputs
- Reduce to an inductive synthesis problem
CEGIS

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

Insert your favorite checker here

{in_i}
CEGIS in Sketch

\( \exists P \forall in \ (in, P \models Spec) \)
∃ c ∀ in \ (in, Sk(c) \vDash Spec)
CEGIS in Sketch

\[ \exists c \forall in \ Q(in, c) \]
CEGIS in Sketch
CEGIS in Sketch

$Q(c, in)$
$Q(c, in)$
CEGIS in Sketch

\[ Q(c, in) \]

Synthesize \rightarrow Check

\[ in \]
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize** → **Check**

\( c \rightarrow \text{in} \)

\( \text{in} \leftarrow \text{in} \)

Diagram showing a graph with nodes A, B, C, D, and edges connecting them.
CEGIS in Sketch

$Q(c, in)$

Synthesize

$Q \ (c, in_0)$

Check

$c$

$in$
CEGIS in Sketch

\[ Q(c, in) \]

\[ Q(c, in_0) \]

Synthesize

Check

\[ c \]

\[ in \]
CEGIS in Sketch

$Q(c, in)$

$Q(c, in_0)$

$\neg Q(c, in_1)$
CEGIS in Sketch

\[ Q(c, \text{in}) \]

Synthesize

\[ Q(c, \text{in}_0) \]

Check
CEGIS in Sketch

$Q(c, in)$

Synthesize

$Q(c, in_0)$

Check

in

c
CEGIS in Sketch

\[ Q(c, in) \]

Synthesize

\[ Q(c, in_0) \]

Check

\[ \neg Q(c, in_2) \]
CEGIS in Sketch

$Q(c, in)$

Synthesize

$Q(c, in_0)$
$Q(c, in_2)$

Check

$c$

$in$
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize**

\[ Q(c, in_0) \]
\[ Q(c, in_2) \]

**Check**

\[ in \]
CEGIS in Sketch

\[ Q(c, in) \]

\[
\begin{align*}
Q(c, in_0) \\
Q(c, in_2)
\end{align*}
\]

\[
\neg Q(c, in_3)
\]
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize**

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

**Check**

\[ c \]

\[ in \]
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize**

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

**Check**

\[ in \]
CEGIS in Sketch

$Q(c, in)$

**Synthesize**

$Q(c, in_0)$

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

**Check**

$\neg Q(c, in_4)$
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize**

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

**Check**

\[ \neg Q(c, in_4) \]
Example

You want to partition $N$ elements over $P$ procs

• How many elements should a processor get?

$N = 18$
$P = 5$
Example

You want to partition \( N \) elements over \( P \) procs

- How many elements should a processor get?

\[ \text{N} = 18 \]
\[ \text{P} = 5 \]

Obvious answer is \( N/P \)
Example

You want to partition $N$ elements over $P$ procs

- How many elements should a processor get?

Obvious answer is $N/P$

Obvious answer is wrong!
Synthesizing a partition function
What do we know?
void partition(int p, int P, int N, ref int ibeg, ref int iend) {
    iend = expr({p, P, N, N/P, N%P}, {{PLUS, TIMES}});
    ibeg = expr({p, P, N, N/P, N%P}, {{PLUS, TIMES}});
}

What do we know?

- The interface to the function we want
void partition(int p, int P, int N, ref int ibeg, ref int iend) {
    if (p < expr({p, P, N, N/P, N%P}, {PLUS, TIMES})) {
        iend = expr({p, P, N, N/P, N%P}, {PLUS, TIMES});
        ibeg = expr({p, P, N, N/P, N%P}, {PLUS, TIMES});
    } else {
        iend = expr({p, P, N, N/P, N%P}, {PLUS, TIMES});
        ibeg = expr({p, P, N, N/P, N%P}, {PLUS, TIMES});
    }
}
void partition(int p, int P, int N, ref int ibeg, ref int iend){
    if(p<
        iend =
        ibeg =
    }else{
        iend =
        ibeg =
    }
}
void partition(int p, int P, int N, ref int ibeg, ref int iend){
    if(p<
        iend =
        ibeg =
    )else{
        iend =
        ibeg =
    }
}

Synthesizing a partition function

What do we know?
• The interface to the function we want
• Not all processors will get the same # of elements
• The kind of expressions we expect
void partition(int p, int P, int N, ref int ibeg, ref int iend){
    if (p<
        iend =
        ibeg =
    )else{
        iend =
        ibeg =
    }
}

Synthesizing a partition function

What do we know?

• The interface to the function we want
• Not all processors will get the same # of elements
• The kind of expressions we expect
Synthesizing a partition function

What do we know?

• The interface to the function we want
• Not all processors will get the same # of elements
• The kind of expressions we expect

```c
void partition(int p, int P, int N, ref int ibeg, ref int iend){
    if(p< expr({p, P, N, N/P, N%P},{PLUS,TIMES})) {
        iend = expr({p, P, N, N/P, N%P},{PLUS,TIMES});
        ibeg = expr({p, P, N, N/P, N%P},{PLUS,TIMES});
    } else{
        iend = expr({p, P, N, N/P, N%P},{PLUS,TIMES});
        ibeg = expr({p, P, N, N/P, N%P},{PLUS,TIMES});
    }
}
```
Synthesizing a partition function

How does the system know what a partition is?
Synthesizing a partition function

How does the system know what a partition is?

```c
harness void testPartition(int p, int N, int P){
    //
}
```
Synthesizing a partition function

How does the system know what a partition is?

```c
harness void testPartition(int p, int N, int P) {
    int ibeg, iend;
    partition(p, P, N, ibeg, iend);
    assert iend - ibeg < (N/P) + 2;
    if (p+1 < P) {
        int ibeg2, iend2;
        partition(p+1, P, N, ibeg2, iend2);
        assert iend == ibeg2;
    }
    if (p==0) { assert ibeg == 0; }
    if (p==P-1) { assert iend == N; }
}
```

- Partitions should be balanced
- Adjacent partitions should match
- First and last partition should go all the way to the ends
Synthesizing a partition function

How does the system know what a partition is?

```c
harness void testPartition(int p, int N, int P){
    if(p>=P || P < 1){ return; }

    int ibeg, iend;
    partition(p, P, N, ibeg, iend);
    assert iend - ibeg < (N/P) + 2;
    if(p+1 < P){
        int ibeg2, iend2;
        partition(p+1, P, N, ibeg2, iend2);
        assert iend == ibeg2;
    }
    if(p==0){ assert ibeg == 0; }
    if(p==P-1){ assert iend == N; }
}
```