Program Synthesis

Arjun Bhamra

2025-09-13

First, some credit

- Big thank you to Stefanos Baziotis for his lecture on Youtube, will link at the end
- Also shout-out Armando Solar-Lezama's PhD thesis on Program Synthesis via Sketching, which introduces the CEGIS algorithm, and is the basis for both Stefanos' lecture and most of this talk.
- (fun fact: Solar-Lezama's thesis reinvigorated this entire subfield of CS apparently)

What is program synthesis?

Program Synthesis correspond to a class of techniques that are able to generate a program from a collection of artifacts that establish semantic and syntactic requirements for the generated code. [2]

Why do we care?

1. Automatic programming for the layperson (FlashFill [3])
2. Creating Database Queries
3. Automating aspects of concurrent programming, such as lock sequencing [1]
4. Synthesized (domain-specific) compiler construction (mlirSynth [4], xDSL-SMT [5])

Research is rife with examples!

Some brief intuition for Sketching

- When we have a programming problem to solve, we don't usually go in blind.
- Instead, we usually have some *intuition* about how to solve the problem; these **partial programs** with a good amount of structure and **holes** which a synthesizer can fill. Call them *sketches*.
- These holes are solved for via **SAT Solvers**, which are computer programs that take in a boolean formula F and decide whether or not F is SAT.
- You don't need to know how SAT Solvers work, just that they are used in synthesis algorithms.

A Motivating Example

```
list reverse(list l){
  if( isEmpty(l) ){
    return l;
  } else {
    node n = popHead(l);
    return append(reverse(l), n);
  }
}
// uses recursion, naive and inefficient
```

```
list reverseEfficient(list l){
  list nl = new list();
  while( ?? ){ ?? }
}
// iterative!
// still large search space?
```

A Motivating Example (cont.)

SAT is in NP, and we believe $\mathrm{P} \neq \mathrm{NP}$. This sucks, but we can make this synthesis practically realizable by adding something else to our sketches; *generators*.

We already have a decent idea of what sort of programs are going to go in our sketch's holes, so why not specify that with a grammar?

```
list reverseEfficient(list l){
  #define LOC {| (l | nl).(head | tail)(.next)? | null |}
  #define COMP {| LOC ( == | != ) LOC |}
  list nl = new list();
  while( COMP ){ ?? }
}
// This is a starting example of a sketch and its generators
```

We also must assert when things are or aren’t correct, either with a naive implementation or some other formal verification logic.

Thus, a sketch needs three main things:

1. A main program
2. Holes
3. Assertions

Now, let’s take a step back and build up our formalism :D

Hoare Logic

- “The goal of Hoare logic is to provide a formal system for reasoning about program correctness” [6]
- We deal in **preconditions** and **postconditions** surrounding a *command*
- Both {pre, post} conditions are *predicates* that a given command relies on for a formal specification
- A **Hoare Triple** is given by $\{A\}\mathrm{cmd}\{B\}$, where:
  1. If a command *cmd* begins execution in a state satisfying A
  2. and *cmd* eventually terminates in some final state F
  3. then F will satisfy assertion B.
- {A, B} are **logical assertions**, they **describe a set of states that satisfy them**

$$\frac{\{A \wedge b\}c_1\{B\}\ \{A \wedge \neg b\}c_2\{B\}}{\{A\}(\text{if } b \text{ then } c_1 \text{ else } c_2)\{B\}}\text{If} \qquad \frac{\{A\}c_1\{B\}\ \{B\}c_2\{C\}}{\{A\}c_1; c_2\{C\}}\text{Seq}$$

Why do we care about Hoare Logic?

Put simply, Hoare Logic provides us a formal system by which to specify assertions that a synthesized program has to satisfy.

Once we attempt to synthesize a program, we must verify that it satisfies our assertions, else add to a list of counterexamples and try again (we will see this idea soon!)

Another Motivating Example

Take the following Hoare Triple: $\{x = 5\}x := x * 2\{x > 0\}$

- Is it correct?
- Is it strict?

What is a stronger postcondition? What is the *strongest* postcondition?

- Formally, if $\{P\}S\{Q\}$ and for all Q such that $\{P\}S\{Q\}, Q \Rightarrow Q$, then Q is the strongest postcondition of S with respect to P (per [6])
- Finding the strongest postcondition is a dual problem to the weakest precondition, and is related to *symbolic execution* (which has its own uses in model checking) [7]
- For more on symbolic execution and analysis, see [8]

Weakest Precondition

If $\{P\}S\{Q\}$ and for all P such that $\{P\}S\{Q\}, P \Rightarrow P$, then P is the weakest precondition wp(S, Q) of S with respect to Q.

Why do we want a weakest precondition?

A stronger condition is satisfied by fewer states, and as such it is **harder to search for a satisfying assignment**. “[A weaker assignment] is less restrictive about input states [...] on which it ensures correctness” [9].

As such, we usually want the weakest such precondition that guarantees that the postconditon holds and the program terminates!

Computing the WPC?

The Weakest Precondition can be easily computed for loop-free programs (as shown in Lecture 18 of [10]), but for programs with loops, this becomes very hard.

Why could this be?

Hint: Consider a while loop, what is a precondition for it? What about after a single iteration - does the precondition change?

Loop Invariants

- They allow us to constrain the valid values that variables can take, and provide some guarantee of "reduction" during each iteration of the loop, that implies eventual termination.
- What we really care about is whether for some program `while b do S` with condition b and body S, there exists a set of loop invariants Inv such that:
  1. $A \Rightarrow \mathrm{Inv}$ (the loop invariants are initially true)
  2. $\{\mathrm{Inv} \;\&\&\; b\}\ S\ \{\mathrm{Inv}\}$ (invariants maintained while loop condition b is valid)
  3. $\{\mathrm{Inv} \;\&\&\; \neg b\} \Rightarrow Q$ (invariant + loop exit condition imply postcondition)

Verification Conditions

- Verification Conditions (VCs) are logical formulas that must hold if the program satisfies its specification, defined by $\mathrm{vc}(\mathrm{cmd}, \mathrm{B})$ (where B is the postcondition as before)
- VCs are valid preconditions ($\{\mathrm{vc}(\mathrm{cmd}, \mathrm{B})\}\ \mathrm{cmd}\ \{\mathrm{B}\}$ is valid) so all we need to do is check that $\mathrm{A} \Rightarrow \mathrm{vc}(\mathrm{cmd}, \mathrm{B})$, and by transitivity we are good!
- However, there is no completeness property here:
  - If $\mathrm{A} \Rightarrow \mathrm{vc}(\mathrm{cmd}, \mathrm{B})$, we know we have satisfied the program's requirements
  - **However**, if $\mathrm{A} \Rightarrow \mathrm{vc}(\mathrm{cmd}, \mathrm{B})$ **does not hold**, then we don't know if the program/triple $\{\mathrm{A}\}\ \mathrm{cmd}\ \{\mathrm{B}\}$ is invalid, or if it is valid but the VC is bad.
  - Thus, the A to vc implication's correctness is crucial
- How do we make good VCs? Do we have to hand-write them or not?

Some observations

Given a set of loop invariants, we can synthesize a verification condition

Without giving each of you psychic damage, the intuiton is roughly that:

1. We know we want to verify some program
2. The program's correctness hinges on the structure of the invariants
3. If we can synthesize the invariants to make the VC valid, we have succeeded!

We are trying to prove that the assertion will be valid for our assumptions (incl. invariants)

As such, we can *hypothesize* that our invariants are a function of all of the variables in scope $\text{inv}(v_1, v_2, ...)$, and then synthesize it by using similar strategies to before (either via sketching or syntax guided synthesis (SyGus) (ask me about this later))

An example of proving the effectiveness of loop invariants

I'm not going to actually go over it because that may take too long, but a good example is § 2 of [6] (starting on page 3).

It is a bit involved but may clarify some of the stuff I've explained.

(The middle of the thesis)

It focuses on understanding deductive vs inductive reasoning, and formalizing the denotational semantics of the Sketch programming language

Reasoning:

- Deductive: Building up formalism from axioms, “bottom up”
- Inductive: Generating formalism from examples, “top down”

Formalizing Semantics (the main ways):

- Denotational: Create mathematical functions called *denotations* that describe the meaning of a language’s expressions (for command c, $[\![c]\!] : \mathrm{State} \to \mathrm{State}'$)
- Operational: Programs are described by how they execute step by step ($\langle P, \sigma \rangle \to \langle P', \sigma' \rangle$)
- Axiomatic: Described by the logical invariants they uphold (like Hoare Logic)

(The middle of the thesis) (cont.)

The reason any of the stuff on the previous slide is important is because it describes the formalism of Armando's work

Your learning it will allow you to read his thesis (which you should do, it should be somewhat approachable after this, if you're interested)

Also, Inductive Synthesis is the idea of generating a program from observations of its behavior (aka, examples)

Counterexample Guided Inductive Synthesis

- The **Bounded Observation Hypothesis** states that for some sketch P, it is possible to find a small set of input examples E that constrains synthesis enough to guarantee that any controls that satisfy the examples $e \in E$ will also satisfy the sketch's resolution equation (this is paraphrased, but essentially it)

- The whole algorithm:
  1. Take in some input examples E, and derive control functions (that encode constraints/concrete program values to fill holes based on $e \in E$)
  2. Try synthesizing a function that satisfies our program correctness conditions
  3. Verify that the given inputs for the holes satisfy the program for **all inputs**, and if it doesn't, add it to the list of counterexamples.

“The generator generates candidate programs drawn from the grammar, and the checker checks the candidates against the spec for correctness. If a candidate satisfies the spec, CEGIS outputs it as the solution. Otherwise the checker asks the generator for more candidates, possibly providing some feedback to the generator.”

– Remy Wang (from [11])

Example to think about

Lets say we want to synthesize $f(x) = 2 * x$ We can start with a random starting candidate and a set of counterexamples E that we want to satisfy.

Then, say the synthesizer generates a candidate $f(x) = 0$.

The verifier will find a counterexample when $x = 1$, so we add that to E, and repeat the process.

Utility

The CEGIS loop's "generator"/synthesizer can be implemented naively with an exhaustive search, but we use SMT solvers because they can actually learn from our counterexamples, giving us some measure of practicality.

Note: We usually define the synthesis process by checking if the *negation* of the condition is SAT. If the negation of a candidate program is satisfiable, then this means the verifier has **found a counterexample** and can send that over for future iterations.

Similarly, if the negation of the condition is UNSAT, this means there **does not exist a counterexample**, and we have generated a program that correctly follows our specification!

Thank you!

Please feel free to ask any question you have, and whether you'd like to hear more talks on this sort of stuff

There are a few papers I'd be happy to talk about that apply this concept, and it would give me a chance to learn more as well

Any other topic requests?

References

1. people.csail.mit.edu/asolar/papers/thesis.pdf
2. https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture1.htm
3. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/popl11-synthesis.pdf
4. https://arxiv.org/pdf/2310.04196
5. github.com/opencompl/xdsl-smt
6. https://www.cs.cmu.edu/~aldrich/courses/654-sp06/notes/3-hoare-notes.pdf
7. https://www.cs.cmu.edu/~15414/s22/lectures/11-post.pdf
8. https://www.cs.cmu.edu/~aldrich/courses/17-355-18sp/notes/notes14-symbolic-execution.pdf

9. https://cecchetti.sites.cs.wisc.edu/cs704/2024fa/notes/lec09-hoare-logic.pdf
10. https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm (**Armando's course notes for Program Synthesis @ MIT**)
11. https://remy.wang/blog/cegis.html