Lecture Notes: Program Synthesis

17-355/17-665/17-819O: Program Analysis (Spring 2018)
Jonathan Aldrich and Claire Le Goues
aldrich@cs.cmu.edu, clegoues@cs.cmu.edu

Note: A complete, if lengthy, resource on inductive program synthesis is the book "Program Synthesis" by Gulwani *et. al* [8]. You need not read the whole thing; I encourage you to investigate the portions of interest to you, and skim as appropriate. I drew many references in this document from there; if you are interested, it contains many more.

1 Program Synthesis Overview

The problem of program synthesis can be expressed as follows:

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

That is, we seek a program P that satisfies some specification φ on all inputs. We take a liberal view of P in discussing synthesis, as a wide variety of artifact types have beeen successfully synthesized (anythign that reads inputs or produces outputs). Beyond (relatively small) program snippets of the expected variety, this includes protocols, interpreters, classifiers, compression algorithms or implementations, scheduling policies, cache coherence protocols for multicore processors. The specification φ is an expression of the user intent, and may be expressed in one of several ways: a formula, a reference implementation, input/output pairs, traces, demonstrations, or a syntactic *sketch*, among other options.

Program synthesis can thus be considered along three dimensions:

(1) Expressing user intent. User intent (or φ in the above) can be expressed in a number of ways, including logical specifications, input/output examples [4] (often with some kind of user- or synthesizer-driven interaction), traces, natural language [3, 7, 13], or full- or partial programs [**?**]. In this latter category lies reference implementations, such as executable specifications (which gives the desired output for a given input) or declarative specifications (which checks whether a given input/output pair is correct). Some synthesis techniques allow for multi-modal specifications, including pre- and post- conditions, safety assertions at arbitrary program points, or partial program templates.

Such specifications can constrain two aspects of the synthesis problem:

- **Observable behavior**, such as an input/output relation, a full executable specification or safety property. This specifies *what* a program should compute.
- **Structural properties**, or internal computation steps. These are often expressed as a sketch or template, but can be further constrained by assertions over the number or variety of operations in a synthesized programs (or number of iterations, number of cache misses, etc,

depending on the synthesis problem in question). Indeed, one of the key principles behind the scaling of many modern synthesis techniques lie in the way they syntactically restrict the space of possible programs, often via a sketch, grammar, or DSL.

.

Note that basically all of the above types of specifications can be translated to constraints in some form or another. Techniques that operate over multiple types of specifications can overcome various challenges that come up over the course of an arbitrary synthesis problem. Different specification types are more suitable for some types of problems than others. Alternatively, trace- or sketch-based specifications can allow a synthesizer to decompese a synthesis problems into intermediate program points.

Question: how many ways can we specify a sorting algorithm?

(2) Search space of possible programs. The search space naturally includes programs, often constructed of subsets of normal program languages. This can include a predefined set of considered operators or control structures, defined as grammars. However, other spaces are considered for various synthesis problems, like logics of various kinds, which can be useful for, e.g., synthesizing graph/tree algorithms.

(3) Search technique. At a high level, there are two general approaches to logical synthesis:

- Deductive (or classic) synthesis (e.g., [15]), which maps a high-level (e.g. logical) specification to an executable implementation. Such approaches are efficient and provably correct: thanks to the semantics-preserving rules, only correct programs are explored. However, they require complete specifications and sufficient axiomatization of the domain. These approaches are classically applied to e.g., controller synthesis.
- Inductive (sometimes called syntax-guided) synthesis, which takes a partial (and often multimodal) specification and constructs a program that satisfies it. These techniques are more flexible in their specification requirements and require no axioms, but often at the cost of lower efficiency and weaker bounded guarantees on the optimality of synthesized code.

Deductive synthesis shares quite a bit in common, conceptually, with compilation: rewriting a specification according to various rules to achieve a new program in at a different level of representation. We will (very) briefly overview Denali [11], a prototypical deductive synthesis techniques, using slides. However, deductive synthesis approaches assume a complete formal specification of the desired user intent was provided. In many cases, this can be as complicated as writing the program itself.

This has motivated new inductive synthesis approaches, towards which considerable modern research energy has been dedicated. This category of techniques lends itself to a wide variety of search strategies, including brute-force or enumerative [1] (you might be surprised!), probabilistic inference/belief propagation [6], or genetic programming [12]. Alternatively, techniques based on logical reasoning delegate the search problem to a constraint solver. We will spend more time on this set of techniques.

2 Inductive Synthesis

Inductive synthesis uses inductive reasoning to construct programs in response to partial specifications. The program is synthesized via a symbolic interpretation of a space of candidates, rather than by deriving the candidate directly. So, to synthesize such a program, we basically only require an interpreter, rather than a sufficient set of derivation axioms. Inductive synthesis is applicable to a variety of problem types, such as string transformation (FlashFill) [5], data extraction/processing/wrangling [4, 19], layout transformation of tables or tree-shaped structures [20], graphics (constructing structured, repetitive drawings) [9, 2], program repair [16, 14] (spoiler alert!), superoptimization [11], and efficient synchronization, among others.

Inductive synthesis consists of several family of approaches; we will overview several prominent examples, without claiming to be complete.

2.1 SKETCH, CEGIS, and SyGuS

SKETCH is a well-known synthesis system that allows programs to provide partial programs (a sketch) that expresses the high-level structure of the intended implementation but leaves holes for low-level implementation details. The synthesizer fills these holes from a finite set of choices, using an approach now known as Counterexample-guided Inductive Synthesis (CEGIS) [**?**, 18]. This well-known synthesis architecture divies the problem into *search* and *verification* components, and uses the output from the latter to refine the specification given to the former.

We have a diagram to illustrate on slides.

Syntax-Guided Synthesis (or SyGuS) formalizes the problem of program synthesis where specification is supplemented with a syntactic template. This defines a search space of possible programs that the synthesizer effectively traverses. Many search strategies exist; two especially well-known strategies are *enumerative search* (which can be remarkably effective, though rarely scales), and *deductive* or *top down* search, which recursively reduces the problem into simpler sub-problems.

2.2 Oracle-guided synthesis

Templates or sketches are often helpful and easy to write. However, they are not always available. Beyond search or enumeration, constraint-based approaches translate a program's specification into a constraint system that is provided to a solver. This can be especially effective if combined with an outer CEGIS loop that provides oracles.

This kind of synthesis can be effective when the properties we care about are relatively easy to verify. For example, imagine we wanted to find a maximum number m in a list l.

Turn to the handout...

Note that instead of proving that a program satisfies a given formula, we can instead disprove its negation, such as:

$$\exists l, m : (P_{max}(l) = m) \wedge (m \notin l \vee \exists x \in l : m < x)$$

If the above is satisfiable, a solver will give us a counterexample, which we can use to strengthen the specificaiton. We can even make this counterexample constructive, so that it provides us an input together with the corresponding correct output m^*:

$$\exists l, m^* : (P_{max}(l) \neq m^*) \wedge (m^* \in l) \wedge (\forall x \in l : m^(*) \geqslant x)$$

This is a much stronger constraint than the original counterexample. This approach was originally introduced for SKETCH, and generalized to oracle-guided inductive synthesis by Jha and Seshia. Different oracles have been developed for this type of synthesis. We will discussed component-based oracle-guided program synthesis in detail, which illustrates the use of distinguishing oracles.

3 Oracle-guided Component-based Program Synthesis

Problem statement and intuition. [1] Given a set of input-output pairs $< \alpha_0, \beta_0 > \ldots < \alpha_n, \beta_n >$ and N components $f_1, \ldots f_n$, the goal is to synthesize a function f out of the components such that $\forall \alpha_i . f(\alpha)$ produces β_i. We achieve this by constructing and solving a set of constraints over f, passing those constraints to an SMT solver, and using a returned satisfying model to reconstruct f. The key idea is that we define a set of *location variables* for each component and inputs and outputs. The synthesis process then reduces to finding values for those location variables, which then tell us which line of the program on which each component should appear. This requires two sets of constraints: one to ensure the program is *well-formed*, and the other that ensures the program encodes the desired *functionality*.

Definitions. We assume for simplicity that each component has a single output, and one or more inputs. The inputs for the i^{th} component, are denoted as $\overrightarrow{\chi}_i$; its output, r_i. Q denotes the set of all input variables from all components; R the set of output variables from all components. Finally, for all variables x, we define a location variable l_x, which denotes where x is defined. L is the set of all location variables:

$$\begin{aligned} Q &:= \textstyle\bigcup_{i=1}^{N} \overrightarrow{\chi}_i \\ R &:= \textstyle\bigcup_{i=1}^{N} r_i \\ L &:= \{l_x | x \in Q \cup R \cup \overrightarrow{\chi} \cup r\} \end{aligned}$$

Well-formedness. ψ_{wfp} denotes the well-formedness constraint. Let $M = |\overrightarrow{\chi}| + N$, where N is the number of available components:

$$\begin{aligned} \psi_{wfp}(L, Q, R) \stackrel{def}{=} & \bigwedge_{x \in Q} (0 \leqslant l_x < M) \ \wedge \bigwedge_{x \in R} (|\overrightarrow{\chi}| \leqslant l_x < M) \ \wedge \\ & \psi_{cons}(L, R) \wedge \psi_{acyc}(L, Q, R) \end{aligned}$$

The first line of that definition says that inputs must be defined before outputs. ψ_{cons} and ψ_{acyc} dictate that there is only one component in each line and that the inputs of each component are defined before they are used, respectively:

$$\begin{aligned} \psi_{cons}(L, R) &\stackrel{def}{=} \bigwedge_{x, y \in R, x \not\equiv y} (l_x \neq l_y) \\ \psi_{acyc}(L, Q, R) &\stackrel{def}{=} \bigwedge_{i=1}^{N} \bigwedge_{x \in \overrightarrow{\chi}_i, y \equiv r_i} l_x < l_y \end{aligned}$$

Functionality. ϕ_{func} denotes the functionality constraint that guarantees that the solution f satisfies the given input-output pairs:

[1] These notes are inspired by Section III.B of Nguyen *et al.*, ICSE 2013 [17] ...which provides a really beautifully clear exposition of the work that originally proposed this type of synthesis in Jha *et al.*, ICSE 2010 [10].

$$\begin{aligned}
\phi_{func}(L,\alpha,\beta) &\stackrel{def}{=} \psi_{conn}(L,\overrightarrow{\chi},r,Q,R) \wedge \phi_{lib}(Q,R) \wedge (\alpha = \overrightarrow{\chi}) \wedge (\beta = r) \\
\psi_{conn}(L,\overrightarrow{\chi},r,Q,R) &\stackrel{def}{=} \bigwedge_{x,y \in Q \cup R \cup \overrightarrow{\chi} \cup \{r\}} (l_x = l_y \Rightarrow x = y) \\
\phi_{lib}(Q,R) &\stackrel{def}{=} (\bigwedge_{i=1}^{N} \phi_i(\overrightarrow{\chi}_i, r_i))
\end{aligned}$$

ψ_{conn} encodes the meaning of the location variables: If two locations are equal, then the values of the variables defined at those locations are also equal. ϕ_{lib} encodes the semantics of the provided basic components, with ϕ_i representing the specification of component f_i. The rest of ϕ_{func} encodes that if the input to the synthesized function is α, the output must be β.

Almost done! ϕ_{func} provides constraints over a single input-output pair α_i, β_i, we still need to generalize it over all n provided pairs $\{< \alpha_i, beta_i > | 1 \leqslant i \leqslant n\}$:

$$\theta \stackrel{def}{=} (\bigwedge_{i=1}^{n} \phi_{func}(L,\alpha_i,\beta_i)) \wedge \psi_{wfp}(L,Q,R)$$

θ collects up all the previous constraints, and says that the synthesized function f should satisfy all input-output pairs and the function has to be well formed.

LVal2Prog. The only real unknowns in all of θ are the values for the location variables L. So, the solver that provides a satisfying assignment to θ is basically giving a valuation of L that we then turn into a constructed program as follows:

Given a valuation of L, `Lval2Prog(L)` converts it to a program as follows: The i^th line of the program is $r_j = f_j(r_{\sigma(1)}, ..., r_{\sigma(\eta)})$ when$l_{r_j} == i$ and $\bigwedge_{k=1}^{\eta} (l_{\chi_j^k} == l_{r_{\sigma(k)}})$, where η is the number of inputs for component f_j and χ_j^k denotes the k^{th} input parameter of component f_j. The program output is produced in line l_r.

Example. Assume we only have one component, +. + has two inputs: χ_+^1 and χ_+^2. The output variable is r_+. Further assume that the desired program f has one input χ (which we call $\mathtt{input}^0$ in the actual program text) and one output r. Given a mapping for location variables of: $\{l_{r_+} \mapsto 1, l_{\chi_+^1} \mapsto 0, \chi_+^2 \mapsto 0, l_r \mapsto 1, l_\chi \mapsto 0\}$, then the program looks like:

$$\begin{array}{ll}
0 & r_0 := \mathtt{input}^0 \\
1 & r_+ := r_0 + r_0 \\
2 & \mathtt{return}\; r_+
\end{array}$$

This occurs because the location of the variables used as input to + are both on the same line (0), which is also the same line as the input to the program (0). l_r, the return variable of the program, is defined on line 1, which is also where the output of the + component is located. (l_{r_+}). We added the return on line 2 as syntactic sugar.

References

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak,

and A. Udupa. Syntax-guided synthesis. In M. Irlbeck, D. A. Peled, and A. Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[2] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. *SIGPLAN Not.*, 51(6):341–354, June 2016.

[3] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, New York, NY, USA, 2016. ACM.

[4] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 9–14, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[5] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.

[6] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 277–289, New York, NY, USA, 2007. ACM.

[7] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 803–814, New York, NY, USA, 2014. ACM.

[8] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[9] B. Hempel and R. Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 379–390, New York, NY, USA, 2016. ACM.

[10] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[11] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002.

[12] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, ATVA '08, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 193–206, New York, NY, USA, 2013. ACM.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[15] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, Mar. 1971.

[16] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering*, ICSE '16, pages 691–701, 2016.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[18] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. *SIGPLAN Not.*, 50(10):107–126, Oct. 2015.

[19] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 343–356, New York, NY, USA, 2016. ACM.

[20] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[21] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. *SIGPLAN Not.*, 51(6):508–521, June 2016.