Scaling Program Synthesis by Exploiting Existing Code

James Bornholt Emina Torlak
University of Washington
{bornholt, emina}@cs.washington.edu

Abstract

Program synthesis automatically produces a program that meets a desired behavioral specification. While synthesis has seen success in a number of domains, interesting applications such as approximate computing and hardware synthesis require more scalability than existing approaches provide. The current approach in synthesis is to achieve scalability by decomposing the problem manually. Inspired by recent success in statistical language models, we propose instead exploiting existing code, using machine learning to guide the synthesis search and automatically decompose the problem.

1. The Need for Scalable Synthesis

Program synthesis is the task of automatically producing a program that meets a desired correctness specification. Search-based synthesis [1, 5, 7, 12, 14, 15] searches for a correct program in a space of candidate implementations. Existing synthesis techniques include brute-force enumeration of the candidate space using dynamic programming [15]; random search with heuristics to explore more fruitful candidates [12]; or formulating the synthesis problem in a logic for an SMT solver to consume [5]. All of these techniques have advantages on particular classes of problems [1], and search-based synthesis has seen success as a programming model in a variety of domains, including low-power spatial computing [9], bulk-synchronous distributed programming [16], and cache coherence protocols [15].

But many promising future applications of synthesis are impeded by the limited scalability of existing techniques:

- Automated synthesis of symbolic execution engines [4] improves the reliability and reach of those tools, but existing work requires significant manual intervention to make the search tractable.
- Our own recent work[1] on applying synthesis to approximate computing [2] focused on approximations of small, manually identified kernels, because the synthesizer could not reason about the entire program.
- Program synthesis could be used to automatically generate hardware implementations from programs. High-level synthesis tools [8] address this problem, but they make limited use of program synthesis techniques due to scalability issues, and instead perform macro expansion that often requires manual intervention.

[1] Currently under submission.

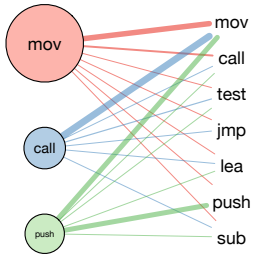

Figure 1. Distribution of common x86 instructions in a WebKit binary. `mov` is by far the most common instruction, and different instructions on the left have different likely following instructions on the right.

The state-of-the-art approach to achieving scalable program synthesis is to manually decompose the problem and specialize synthesis to it. For example, *sketching* [13] constrains the space of candidate solutions with a partial implementation, which decomposes the synthesis task into smaller *holes* to be filled by a synthesizer. Designing a tractable sketch requires both creativity and understanding of the underlying search. In this paper we canvas potential techniques to *automate* problem decomposition for scalable program synthesis by exploiting machine learning. An automated approach would make synthesis much easier to apply to new problem domains such as those above.

2. Exploiting Properties of Programs

We highlight the well-known observation [6] that the operations or instructions in a program tend not to be uniformly distributed: at a given program location, not each instruction is equally likely to appear. This is true both because some instructions are simply more common than others, and because instructions often appear in repeated patterns and idioms.

Figure 1 illustrates this effect by analyzing x86 instructions in a WebKit binary. The left hand side shows the three most common instructions in the binary, scaled by frequency; `mov` accounts for 39% of the instructions in the binary. The right hand side shows several common instructions in the binary, and the weight of the edges between the sides show

how frequently an instruction on the right immediately follows the instruction on the left. For example, 62% of `call` instructions (on the left) are followed immediately by a `mov` instruction (on the right), and 46% of `push` instructions are followed immediately by another `push`.

This data points to common idioms at the assembly level. The same patterns can be seen in higher-level languages; for example, a call to `malloc` is likely to be followed by an error-checking conditional, and quite unlikely to be immediately followed by a `free`. Similarly, some functions are much more likely to appear in a program than others: `malloc` is much more common than `tgamma`, even though both are members of the C standard library.

Today's program synthesis algorithms generally do not exploit this rich structure. Stochastic search, as implemented by Schkufza et al. [12], is equally likely to mutate an instruction to any other instruction in the same equivalence class. One form of solver-based synthesis uses a bag of components to complete an SSA program [5], but will consider (or rule out by deduction) every combination of instructions without preferring those most likely to succeed. Some existing work applies simple probabilistic models to enumerative search [3], but we believe more rich and general solutions are possible.

3. Decomposing Synthesis with "Big Code"

We propose the use of machine learning models to guide and inform search-based program synthesis. We are inspired by the recent success of statistical language models for code completion [10] and inferring program properties [11]. Rather than generating programs from statistical models, however, we propose augmenting existing program synthesis techniques with statistical observations from real-world code.

Component-Based Synthesis. Gulwani et al.'s algorithm for synthesizing loop-free programs is based on a bag of *components* that an SMT solver tries to connect together into a loop-free SSA-form program [5]. The only requirement for these components is that they have a logical encoding of their input-output behavior. Current implementations use low-level bitvector operations as the bag of components, which limits their applicability.

Common program idioms are longer than a single instruction. We propose mining existing code bases for common idioms, computing symbolic summaries of common code blocks, and using these summaries as components. Using larger blocks as components for synthesis will allow the search to scale to longer programs with more complex behavior. Moreover, using larger components allows for a decomposed hierarchical synthesis approach, where the individual components can be re-synthesized in isolation to find an optimal implementation. Such an approach is particularly well suited to synthesizing hardware, where low-level building blocks (such as adders) have complex implementations but simple logical semantics.

Probabilistic Search Heuristics. Statistical models for programs can also tell us which instructions are likely to follow other instructions (as in Figure 1). We propose exploiting statistical language models to guide search-based synthesis. For example, if the synthesizer is currently exploring a partial solution in which instruction i is a `mov`, our language model can guide the exploration of potential instructions at position $i+1$. More generally, language models can tell us which instructions often occur together, so a partial implementation can guide the search towards more relevant instructions.

Mining Sketches for Structure. Sketch-based synthesis starts with a partial implementation of the desired program, where *holes* indicate missing expressions to be filled in by the synthesizer [13]. Current applications of sketching generally require sketches to be specified manually, expecting programmers to determine the problem decomposition. For example, work on synthesizing symbolic execution engines [4] required manual intervention to define a set of templates that could represent (a subset of) x86 instructions.

We propose mining existing code bases for sketches. In particular, current synthesis techniques often can only synthesize loop-free programs with simple control flow. If existing code bases expose common control-flow idioms, we can extract the structure of those programs as a sketch, which can be filled in by an existing sketch-based synthesizer.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[3] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

[4] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *PLDI*, 2012.

[5] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[6] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, 2012.

[7] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[8] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. IEEE*, 78, 1990.

[9] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[10] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[11] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *POPL*, 2015.

[12] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[13] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[14] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[15] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *PLDI*, 2013.

[16] Z. Xu, S. Kamil, and A. Solar-Lezama. MSL: A synthesis enabled language for distributed implementations. In *SC*, 2014.