MuSynth: Program Synthesis via Code Reuse and Code Manipulation

Vineeth Kashyap(✉), Rebecca Swords, Eric Schulte, and David Melski

GrammaTech, Inc., Ithaca, NY 14850, USA
{vkashyap,rswords,eschulte,melski}@grammatech.com

Abstract. MuSynth takes a draft C program with "holes", a test suite, and optional simple hints—that together specify a desired functionality—and performs program synthesis to auto-complete the holes. First, MuSynth leverages a similar-code-search engine to find potential "donor" code (similar to the required functionality) from a corpus. Second, MuSynth applies various synthesis mutations in an evolutionary loop to find and modify the donor code snippets to fit the input context and produce the expected functionality. This paper focuses on the latter, and our preliminary evaluation shows that MuSynth's combination of type-based heuristics, simple hints, and evolutionary search are each useful for efficient program synthesis.

Keywords: Program synthesis · Evolutionary computation · Code reuse · Big code

1 Introduction

Software developers have collectively written an enormous amount of code. Availability of such "big code" in searchable archives has spurred recent research [12, 13], with the overarching theme of leveraging existing code to improve developer productivity. In this work, we use "big code" for program synthesis, to automatically generate programs that meet developer's requirements [7].

Code reuse [5] is widespread as it aids rapid prototyping with limited resources. It includes both as-is reuse and reuse involving code modification to fit a new context. As an example of the latter, to reduce time-to-market, developers adapt existing code to run on embedded devices with constrained resources—it may be infeasible to load whole image processing libraries, but code snippets implementing specific functionality from these libraries could be practically re-used and customized for the embedded device.

In many software development scenarios, the functionality that a developer is attempting to create already exists somewhere else, perhaps with minor differences. MuSynth is an automated program synthesis engine that (1) uses

This research was supported by DARPA MUSE award #FA8750-14-2-0270. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

T. Menzies and J. Petke (Eds.): SSBSE 2017, LNCS 10452, pp. 117–123, 2017.
DOI: 10.1007/978-3-319-66299-2_8

partial specifications of the functionality being developed, (2) searches a large corpus for "donor" code that implements a similar functionality, and (3) reuses and modifies the donor code in the developer's context to produce desired functionality. In this paper, we focus on a research question regarding (3): given a partial specification via developer-provided tests and some donor code that potentially implements the functionality, how can we automatically manipulate the donor code to incorporate it into the developer's context? MuSynth combines type-based heuristics, developer hints, and evolutionary search to address this research question. Based on our preliminary results, we believe MuSynth is a novel and promising approach to program synthesis that can be effectively combined with traditional logic-based approaches [3] in the future.

2 Related Work

Program synthesis from partial programs. Syntax-guided synthesis [3] uses partial programs (sketches) and user specifications as input to generate programs. Specifications are logical predicates (pre- and post-conditions) describing the desired behavior of the program. Instead of using purely logical techniques to synthesize holes, our work exploits code reuse and evolutionary search.

Evolutionary search in program repair. Evolutionary search has been successful in the related field of program repair [10]. Although program repair can be viewed as synthesis of bug fixes, unlike program synthesis, program repairs are not expected to generate new functionality.

Program synthesis based on existing code. Recent techniques [4,12] have trained deep neural networks on existing code and used the generated models in program synthesis. MuSynth can augment these techniques. Program splicing [11] uses relevant donor code for synthesis but, unlike MuSynth, performs an exhaustive enumerative search over unmodified (except for variable renaming) donor code.

Evolutionary search and program synthesis. Evolutionary program sketching [6] modifies sketches until traditional techniques [3] can fill holes. Unlike MuSynth, the holes they consider are very simple, and can be filled only with constants or variables. Katz et al. [9] use genetic programming guided by model checking for program synthesis. Whereas their approach requires a formal specification of the functionality being synthesized, MuSynth works with partial specifications.

3 MuSynth Overview

Figure 1 provides an overview of MuSynth. It uses a similar-code-search engine to identify relevant donor code from a large corpus that may implement the required functionality. These donors are then mutated to fit the developer's problem context. The mutations are guided by evolutionary search. This paper

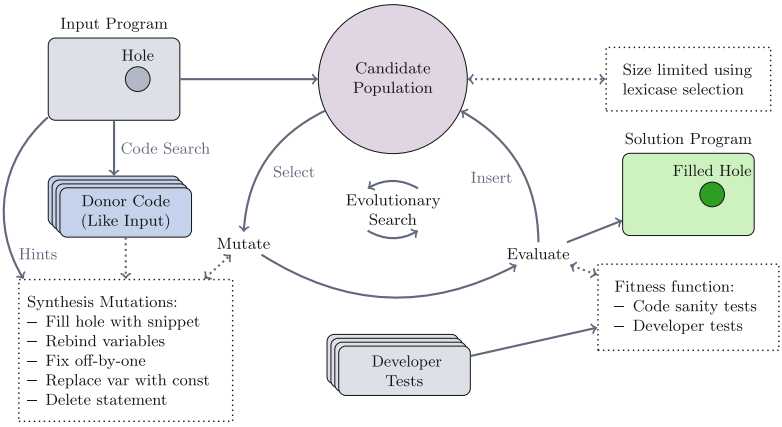

Fig. 1. MuSynth architecture. Gray boxes are the input, green box is the output. (Color figure online)

focuses on the donor code mutations and the evolutionary search in MuSynth, which are built on top of Clang [2] and Software Evolution Library [14].

The developer provides a draft program containing "holes" along with test cases and optional hints that specify the expected functionality of the completed program. MuSynth synthesizes code to appear in the holes and does not modify code outside of the holes. The program context surrounding the holes drives the search for similar functionality (i. e., the donor code) by using Source Forager (SF) [8], a similar-code-search engine. SF takes a C procedure as input—optionally with holes—and returns a list of C procedures from a large corpus that are the most input-similar and potentially relevant for subsequent synthesis. SF employs multiple code features from the surrounding context for code search, such as natural language (comments, variable and function names), abstractions of ASTs (Abstract Syntax Trees), types used and operations performed. Currently, SF can search over a million procedures in under two seconds.

MuSynth maintains a candidate population of program variants: each variant is derived by "filling" the holes in the draft program. At each step of the evolution, (1) a variant is selected, (2) a synthesis mutation is applied to the variant, (3) the variant is evaluated for fitness by compiling and running developer tests and various code sanity tests (i. e., tests finish within reasonable resource usage and time limits, and do not exit/abort early), and (4) based on the results of the evaluation, the mutated variant either re-enters population, or if good enough, is presented as the solution. MuSynth uses lexicase selection [15] over test cases and limits the size of the population to a pre-specified maximum. Evolutionary steps occur in parallel across threads. For successful synthesis, MuSynth requires the code search results to contain at least one relevant donor procedure, high-quality developer tests, and a correct (i. e., feasible to solve) draft program.

Figure 2 shows a simplified example sequence of synthesis mutations. The draft program declares an uninitialized array, a loop to iterate over the array,

leaving a hole to zero-initialize the array. MuSynth always applies the **fill** mutation to any empty hole. The fill mutation first finds a snippet of donor code: various AST subtrees are extracted from the donor code procedures, and one of them is randomly selected, biased by subtree size (smaller is preferred) and the hole position (e. g., if the hole is inside a loop, donor code AST subtrees within loops are preferred). Variables in donor code are then mapped to the ones in the draft code. The number of such possible mappings is typically large, and to reduce this number, MuSynth uses type-based heuristics, i. e., type-compatibility checks between donor and draft variables. E.g, the types "array of `short`" and "pointer to `char`" are considered compatible, because the same operations (e. g., array indexing and bit shifting) can be applied on variables with either type. These type-based heuristics reduce the number of non-compiling variants. The developer can provide simple optional hints to MuSynth, such as (1) the superset of variables expected to be used in synthesized code, and (2) the subset of the variables that must be modified by the synthesized code. These hints help further reduce the number of possible mappings. Here, fill mutation maps the donor variables `x`, `index` to draft variables `array`, `j` respectively.

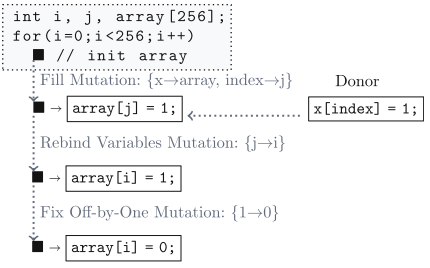

Fig. 2. Example sequence of mutations.

A **rebind** mutation attempts to find an alternate variable mapping for the hole, further exploring potential variable bindings. Here, `j` is renamed to `i` (of compatible type), and `array` is left unchanged. A **fix off-by-one** replaces a constant in the hole with an off-by-one constant: here, `1` is replaced by `0`. Finally, this change creates the correct zero-initialization. Other mutations frequently used by MuSynth are: **replace variable with a constant**, **delete statement**, **refill** (which throws away the current contents of the hole and refills with new donor snippet), and **insert new statement** (insert a new donor statement at random within the hole).

4 Evaluation

Our evaluation goal is to understand the effectiveness of MuSynth on a benchmark suite [1] of program synthesis challenges, openly available for review. The benchmarks consist of 3 algorithms: (1) image contrast enhancement using histogram equalization, (2) binary search, and (3) insertion sort. For each algorithm, we found an existing implementation online, removed code at different program points to create holes, and wrote test cases to specify the expected functionality. As it is not the focus of the paper, we cached the results (4–5 similar procedures [1] per algorithm) of running SF on these benchmarks—they constitute the donor code used in synthesis. The time taken by SF is *not* included in the reported times below. Table 1 summarizes our evaluation. Column 1 lists the

various benchmarks—each is a draft program with a hole of different size and program location. Column 2 indicates a proxy for the expected complexity of the hole as a pair of numbers: (1) the number of n-ary operations expected to be in the hole (such as assignment, array indexing, bit shifting), (2) the number of variables expected to be used in the hole. These numbers were computed based on the original code that was replaced with corresponding holes.

We run MuSynth under 4 configurations to evaluate the research question outlined in Sect. 1. Evolutionary search is used by EVO$^{\text{NONE}}$, EVO$^{\text{SOME}}$, and EVO$^{\text{ALL}}$. EVO$^{\text{NONE}}$ runs without any hints from the developer. EVO$^{\text{SOME}}$ is provided the superset of the variables expected in the holes. In addition to these hints, EVO$^{\text{ALL}}$ is provided the subset of the variables that must be modified by the hole. RAND$^{\text{ALL}}$ is the same as EVO$^{\text{ALL}}$, except that it uses random search instead (i. e., unguided search that always starts with a fill mutation, and then randomly applies a synthesis mutation, but restarts when uncompilable variants are produced). All 4 configurations use type-based heuristics—without which most benchmarks time out—and the same probabilities for all the synthesis mutations. An exhaustive enumerative search of the infinite solution space requires specifying an order for applying the synthesis mutations and variable bindings. Instead of hand-picking one such ordering, we avoid bias by using RAND$^{\text{ALL}}$ for comparison. The rightmost 4 columns in Table 1 provide the time taken (in seconds) for running MuSynth on different benchmarks until a correct solution is found. We run each experiment 10 times and the median time is reported. Experiments were run on an AMD ×64 machine (2.6 GHz, 4 cores, 16 GB main memory), with a population of maximum size 1000, 4 threads, and a 30 min timeout (∞ is used to indicate timeouts). The hole synthesized by running EVO$^{\text{ALL}}$ on the benchmark contrast-enhance-4 is shown in Fig. 3 as an example of MuSynth's synthesis capabilities.

Table 1. Summary of evaluation results (fastest time highlighted in boxes).

Benchmark	Complexity	EVO$^{\text{NONE}}$	EVO$^{\text{SOME}}$	EVO$^{\text{ALL}}$	RAND$^{\text{ALL}}$
contrast-enhance-0	(2, 2)	239.97	5.18	4.88	4.40
contrast-enhance-1	(4, 4)	141.75	28.32	4.88	27.68
contrast-enhance-2	(7, 5)	1792.71	28.93	4.91	5.23
contrast-enhance-3	(10, 5)	∞	200.11	80.56	17.77
contrast-enhance-4	(10, 7)	∞	∞	576.17	∞
binary-search-0	(11, 5)	46.80	69.45	37.24	∞
binary-search-1	(8, 5)	661.06	72.46	12.52	15.05
binary-search-2	(12, 5)	26.32	11.35	12.91	10.97
insertion-sort-0	(16, 4)	19.66	9.65	9.74	9.81
insertion-sort-1	(5, 2)	34.70	13.74	8.18	14.70
insertion-sort-2	(19, 5)	4.96	4.43	4.26	3.55
insertion-sort-3	(4, 2)	5.27	5.66	5.16	3.93

```
for (i = 0; i < L; i++) {
  foo = ((unsigned long)histogram[i] << N);
  cdf += 255 * (foo/(unsigned long)pixels);
  gray_level_mapping[i] = ((unsigned long)(cdf>>N)) & 0xff;
}
```

Fig. 3. Synthesized code (indicated in red) for contrast-enhance-4, which required, among other things, finding the correct name bindings for seven variables and macros, and fixing an off-by-one error. The surrounding context has been elided for space. The donor code that gets adapted here uses a different image representation format and different amount of bit shifting than the synthesized code. This result shows that donor code can be adapted even when it uses different data structures or contains errors. (Color figure online)

Table 1 shows that in almost all cases, EVO^{ALL} is either the fastest, or is close to the fastest. EVO^{ALL} is also the only one to successfully complete (i. e., passes all test cases, with solutions manually verified afterwards) on all benchmarks within the time limit. This result indicates that *developer hints can significantly reduce synthesis time.* RAND^{ALL} is the fastest in a few cases because of the low overhead (i. e., no population to maintain). However, it times out in cases that require sequencing of multiple synthesis mutations. This result indicates that *evolutionary search is useful in navigating a large search space of synthesis mutations.* With these results, we believe that type-based heuristics, developer hints, and evolutionary search contributes to efficient program synthesis.

References

1. Program Synthesis Challenge Benchmark. https://github.com/ssbse-2017-submission/synthesis-challenges
2. The Clang Project. https://clang.llvm.org/
3. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. IEEE (2013)
4. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to Write Programs. ArXiv e-prints., November 2016
5. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In: FSE. ACM (2014)
6. Błądek, I., Krawiec, K.: Evolutionary program sketching. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 3–18. Springer, Cham (2017). doi:10.1007/978-3-319-55696-3_1
7. Gulwani, S.: Dimensions in program synthesis. In: PPDP. ACM (2010)
8. Kashyap, V., Brown, D.B., Liblit, B., Melski, D., Reps, T.: Source Forager: A Search Engine for Similar Source Code. ArXiv e-prints (2017)
9. Katz, G., Peled, D.A.: Synthesis of parametric programs using genetic programming and model checking. In: INFINITY (2013)
10. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)

11. Lu, Y., Chaudhuri, S., Jermaine, C., Melski, D.: Data-Driven Program Completion. ArXiv e-prints, May 2017
12. Murali, V., Chaudhuri, S., Jermaine, C.: Bayesian Sketch Learning for Program Synthesis. ArXiv e-prints (2017)
13. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "Big Code". In: POPL. ACM (2015)
14. Schulte, E.: Neutral networks of real-world programs and their application to automated software evolution. Ph.D. thesis, University of New Mexico, Albuquerque, USA, July 2014. https://cs.unm.edu/~eschulte/dissertation
15. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: GECCO. ACM (2012)