NomNom: Explanatory Function Names for Program Synthesizers

Amirmohammad Nazari
nazaria@usc.edu
University of Southern California
USA

Souti Chattopadhyay
schattop@usc.edu
University of Southern California
USA

Swabha Swayamdipta
swabhas@usc.edu
University of Southern California
USA

Mukund Raghothaman
raghotha@usc.edu
University of Southern California
USA

ABSTRACT

Despite great advances in program synthesis techniques, they remain algorithmic black boxes. Although they guarantee that when synthesis is successful, the implementation satisfies the specification, they provide no additional information regarding how the implementation works or the manner in which the specification is realized. One possibility to answer these questions is to use large language models to construct human-readable explanations. Unfortunately, experiments reveal that LLMs frequently produce nonsensical or misleading explanations when applied to the unidiomatic code produced by program synthesizers. In this paper, we develop an approach to reliably augment the implementation with explanatory names. Experiments and user studies indicate that these names help users in understanding synthesized implementations.

ACM Reference Format:
Amirmohammad Nazari, Souti Chattopadhyay, Swabha Swayamdipta, and Mukund Raghothaman. 2024. NomNom: Explanatory Function Names for Program Synthesizers. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643529

1 INTRODUCTION

The promise of allowing developers to express their intent in more flexible ways has led to an enormous rise of interest in program synthesis over the last fifteen years [Gulwani et al. 2017]. Sophisticated algorithms have been developed that can synthesize non-trivial pieces of code [Alur et al. 2018]. As an example, we invite the reader to inspect the following program $f(l)$, produced by DreamCoder, a recent state-of-the-art program synthesizer [Ellis et al. 2021]:

$$f(l) = \texttt{map}\ (\lambda n.\ g_1(l, 1+n))\ (\texttt{range}(\texttt{len}(l))), \text{ where}$$
$$g_1(l,n) = g_2(\texttt{filter}\ (\lambda z.\ n > \texttt{len}\ (\texttt{filter}\ (\lambda u.\ z > u)\ l))\ l), \text{ and}$$
$$g_2(l) = \texttt{hd}\ (\texttt{filter}\ (\lambda y.\ \texttt{isnil}\ (\texttt{filter}\ (\lambda z.\ z > y)\ l))\ l).$$

This implementation was produced in response to a specification for a program that sorts a list of numbers. A collection of input-output examples was provided, which included, among others:

$$f(\texttt{[9; 2; 7; 1]}) = \texttt{[1; 2; 7; 9]}. \quad (1)$$

It is our contention that difficulty in understanding complicated implementations such as these hinders the practical adoption of program synthesizers. In fact, [Ellis et al. 2021] itself includes helpful comments outlining the purpose of different functions: $f(l)$ sorts the input list l, $g_1(l,n)$ computes the n-th smallest element of l, and $g_2(l)$ computes the largest value in l.

Can we automatically obtain descriptive function names such as these, perhaps by use of a large language model? Unfortunately, because of the unidiomatic code produced by synthesis engines, straightforward use of an LLM leads to poorly chosen function names. For our example, `text-davinci-003` suggests the names "`largestSmallestIndices`", "`findNearestNumber`" and "`getFirstItem MinThanArgumentValue`" for the functions f, g_1 and g_2 respectively.

In this paper, we explore how a combination of two simple ideas can result in significantly more helpful names which programmers can use to understand synthesized implementations.

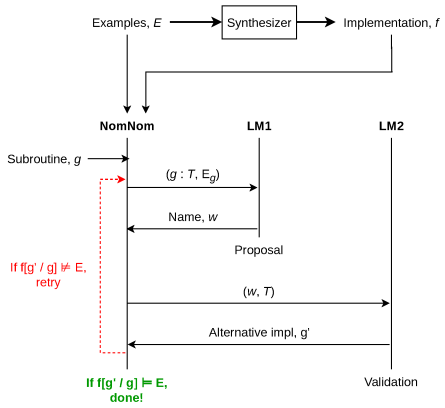
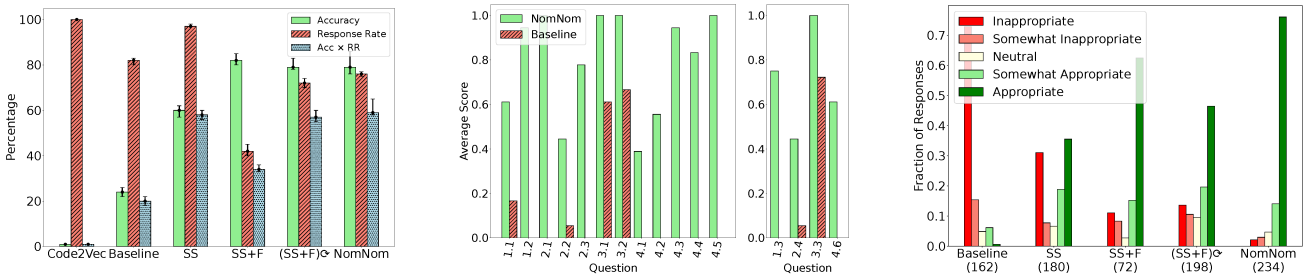


Figure 1: Overall architecture of our system, NomNom. We begin with a specification-implementation pair, (E, f), and a subroutine g of interest. The system alternates between querying a first LLM to obtain candidate names w for g and validating w by using a second language model to resynthesize an alternative implementation.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04.
https://doi.org/10.1145/3639478.3643529



(a) (b) (c)

Figure 2: (2a) Effectiveness of different name generation approaches on a larger dataset. (2b) Accuracy of participant responses when explaining how implementations worked. In the group of questions on the left, we asked participants to identify the purpose of different functions, while in the second group of questions, we asked for a more global explanation of how the implementation worked. (2c) Distribution of participant preferences among function names produced by different algorithms.

2 IDEA 1: AUGMENTING LLM PROMPTS

As a first step, we augment the prompt supplied to the LLM with hints that explain the purpose of each function. Informally, we simply instrument each function with `print` statements that log the inputs flowing into and outputs emerging out of each function:

$$g_2'(l) = \texttt{let}\ ans = g_2(l);\ \texttt{print}\ l \mapsto ans;\ \texttt{return}\ ans.$$

Upon testing the implementation with the monitoring instrumentation enabled, one finds that g_2 performs the mapping:

$$\texttt{[1]} \mapsto 1, \texttt{[2; 1]} \mapsto 2, \texttt{[2; 7; 1]} \mapsto 7, \text{ and } \texttt{[9; 2; 7; 1]} \mapsto 9, \quad (2)$$

which immediately suggests that $g_2(l)$ evaluates to the largest element of the list l. In addition, the original function $g_2(l)$ can be replaced with any implementation $g_2''(l)$ that satisfies Equation 2 without affecting the fact that the original program $f(l)$ satisfies the specification of Equation 1.

Monitoring code in this manner is closely related to the idea of subspecifications, a recent concept introduced by [Nazari et al. 2023] in order to locally explain programs produced by program synthesizers. The chief difference between the two ideas is that subspecifications are both necessary and sufficient conditions for the global implementation to work, while the "*tight*" subspecs in Equation 2 are only sufficient but not always necessary conditions. In other words, subspecs describe what a function is *supposed to do*, while tight subspecs describe what the function *actually does*.

Such hints increase the accuracy of the backbone LLM from 24% to 60%, as shown in Figure 2a, and the system is able to correctly suggest the name "`findLargestElement`" for the function g_2.

3 IDEA 2: ALGORITHMIC SANITY CHECKS

Our second insight is that when a function is appropriately named, that name can be used to substantially recover the original implementation. For example, given the function name `findLargestElement` and its type, GPT-3.5 suggests the implementation:

$$\lambda\, l.\ \texttt{fold}\ (\lambda\, i.\ \lambda\, j.\ \texttt{if}\ i > j\ \texttt{then}\ i\ \texttt{else}\ j)\ (\texttt{hd}\ l)\ (\texttt{tl}\ l).$$

Observe that this new function continues to satisfy the same tight subspec in Equation 2. This suggests that `findLargestElement` is indeed an appropriate name for the subroutine g_2. Although such checks do not guarantee the appropriateness of names, they indicate at least some degree of internal consistency, which we can use to detect and filter out inappropriate function names.

This further boosts the accuracy of the system from 60% to 82% respectively. Although it leads to a drop in the response rate, we can exploit non-determinism in LLM responses, and give it multiple chances to suggest function names that pass the filter. Together with some other optimizations, our complete system achieves an accuracy of 79% while answering 77% of all questions that we ask.

4 ARE OUR NAMES HELPFUL TO USERS?

In order to determine whether names help users in understanding implementations, we chose four programs and asked a group of 18 student programmers questions about them. Each participant attempted some tasks using names produced by the baseline language model, and other tasks with names produced by our synthesizer, NomNom. As one can see in Figure 2b, users clearly achieved better understanding of the code when using names produced by our tool. We then asked another group of 18 student programmers to rate their preference among names produced by different tools. As Figure 2c, there is uniformly increasing preference among participants for names produced by NomNom.

ACKNOWLEDGMENTS

This research was supported in part by NSF grants CCF-2146518, CCF-2124431, and CCF-2107261.

REFERENCES

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (nov 2018), 84–93. https://doi.org/10.1145/3208071

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. ACM, 835–850.

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

Amirmohammad Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. 2023. Explainable Program Synthesis by Localizing Specifications. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 298 (oct 2023), 25 pages. https://doi.org/10.1145/3622874