Generating Function Names to Improve Comprehension of Synthesized Programs

Amirmohammad Nazari*, Swabha Swayamdipta*, Souti Chattopadhyay* and Mukund Raghothaman*
*University of Southern California
Los Angeles, CA, USA
{nazaria, swabhas, schattop, raghotha}@usc.edu

***Abstract*—The hope of allowing programmers to more freely express themselves has led to a proliferation of program synthesis techniques. These tools automatically derive implementations from high-level specifications of user intent. These specifications may take the form of logical formulas, demonstrations, or input-output examples. Synthesizers guarantee that when synthesis is successful, the implementation satisfies the specification. However, they provide no additional information regarding how the implementation works or the manner in which the specification is realized. As a result, they remain algorithmic black boxes which are prone to producing unidiomatic code with procedurally generated identifier names, like `x1`, `x2`, etc. As a result, complicated implementations produced by modern program synthesizers are becoming increasingly hard to understand.**

One solution to this comprehensibility problem is to produce meaningful identifier names for its variables, functions, etc. While large language models (LLMs) suggest a simple way to obtain human-readable names, our experiments reveal that LLMs frequently produce nonsensical or misleading names when applied to code emitted by program synthesizers.

In this paper, we develop an approach to reliably augment the implementation with explanatory names: We recover fine-grained input-output data from the synthesis algorithm to enhance the prompt supplied to the LLM and use a combination of a program verifier and a second language model to validate the proposed names before presenting them to the user. Together, these techniques improve the accuracy of the proposed names from 24% to 79%. A two-phase user study indicates that users significantly prefer the names produced by our technique, and that the proposed names greatly help users in understanding synthesized implementations.

***Index Terms*—Program synthesis, function name generation, large language models, inter-LLM validation**

I. Introduction

The last twenty years have seen an explosion of work in program synthesis [1], [2] with the hope of aiding programmers to freely express their intent. This work has been targeted both towards the underlying algorithms [3]–[9], and their applications [10]–[18].

However, there has been comparatively less attention given to the problem of helping users *understand* this synthesized code. Synthesizers frequently produce unidiomatic code, and do not provide intuitive identifier names, comments, or other hints to help the user understand how the implementation works. In our user study, we asked 18 student programmers to examine four programs produced by DreamCoder [19], a recent state-of-the-art program synthesizer, and discovered that the participants only received an average score of 23% when asked simple questions about these programs.

Although there is research on requesting additional guidance from the user during the synthesis process, this has primarily been oriented either towards accelerating the synthesis process itself [20], or in reducing the number of examples needed to identify the target program [21]. Notably, these approaches do not directly help the user in understanding how the program works and whether it accurately realizes their intent.

We contend that the unintuitive code produced by many program synthesizers reduces users' confidence in using the code, and inhibits the impact of program synthesis technologies. Our focus in this paper is on generating explanatory names for intermediate functions in synthesized code, with the ultimate goal of improving user understanding of code.

Recent breakthroughs in large language models [22]–[26] suggest their possible application to our problem. Unfortunately, as we will see in Section IV, implementations produced by program synthesizers are unidiomatic, leading to poor quality names being suggested by the language model / backend tool. This issue also affects state-of-the-art systems for automatically deriving function names [27]–[29].

Our primary contribution is a novel two-step technique to obtain candidate function names from a first LLM, and use a combination of a second LLM and a program verifier to validate these generated names before presenting them to the user. In our user study, names produced by our technique increased the average score of participants from 23% to 81% when asked questions about the code. Across a set of 144 implementations produced by DreamCoder containing names written by a human expert, our system produced names with an accuracy of 79% compared to the baseline (LLM only) accuracy of 24%.

In a second study, we asked another group of 18 programmers to rate names produced by different name generation algorithms on a five-point Likert scale. Participants strongly prefered names produced by our system with 76% of responses marking these names as appropriate, while only 2% responses found the names generated by the baseline LLM to be appropriate.

This paper is an example of how generative AI can be combined with formal techniques in programmer assistance tools. By generating explanatory names for functions in synthesized programs, we hope to help users understand automatically generated code, and thereby improve the adoption of program synthesis technology.

```
def g2(x2):
  def g21(x21):
    def g22(x22):
      return x21 < x22
    return len(list(filter(g22, x2))) == 0
  return list(filter(g21, x2))[0]

def g1(x1):
  def g11(x11):
    def g12(x12):
      def g13(x13):
        return x12 > x13
      return x11 > len(list(filter(g13, x1)))
    return g2(list(filter(g12, x1)))
  return g11

def f(x1):
  def f1(x11):
    return g1(x1)(x11 + 1)
  return list(map(f1, range(len(x1))))
```

Fig. 1: Program produced by DreamCoder to sort a list of numbers. The top-level function is f. Equation 1 is an excerpt of the specification given to the synthesizer. We transliterated this program into Python from the original lambda-expression which may be found in the supplementary material.

II. Overview and Motivating Example

Consider a user who wants a program that sorts a list of numbers. They may describe their intent using input-output examples such as the following:

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

They may then realize this intent using any of a number of inductive program synthesizers [7], [30]–[32].

In this paper, for the sake of concreteness, we focus on programs synthesized using DreamCoder [19]. Synthesis using DreamCoder runs in two phases: in the first (offline) phase, the system uses a corpus of synthesis tasks to construct a library of reusable components (i.e., functions) which it then uses to more rapidly discharge the provided specification in the subsequent (online) synthesis phase.

We adapt the specification in Equation 1 from Figure 1B of [19]. In response, it produces a lambda term which may be transliterated into the Python code of Figure 1. Notice that the program uses non-trivial language features such as higher-order functions and that its subroutines have uninformative sequentially-generated names (such as `g1`, `g2`, ...). It is therefore difficult to understand how the program works, or even confirm that it always sorts the provided list of numbers.

We also remark that the top-level auxiliary functions, `g1` and `g2`, correspond to reusable components discovered by DreamCoder from the training data. Because the synthesizer concluded that they are useful across a range of tasks, it appears plausible that they perform some high-level conceptually salient operations over lists. Several recent program synthesizers, including Babble [33], Enumo [34] and Stitch [35], similarly learn libraries of reusable components / rewrite rules.

Upon reflecting on this program, one may conclude that invoking the function `g1(l)(n)` produces the `n`-th smallest element of the list `l`, and that the function `g2` returns the largest element of the list `x2` that it accepts as input. In fact, in the original example of [19], the authors manually add expository comments describing the behavior of these intermediate functions. In this section, we provide an overview of our system NomNom: it accepts as input a specification-implementation pair (φ, f) such that f satisfies φ, and uses an LLM to algorithmically produce names for each subroutine g that appears in f.

A. The Baseline LLM

As a baseline, one may request an LLM, such as one from the GPT family, to provide a name for each function g in question. Each prompt includes the body and type of the function $g : T$ being named, and (recursively) any auxiliary functions in the call graph rooted at g. We provide the baseline prompt templates in the supplementary material.

However, when using GPT-3.5,[1] it fails to produce appropriate names for any of the functions in Figure 1. As an example, it suggests the name "`largestSmallestIndices`" for the top-level function f, and "`findNearestNumber`" and "`getFirstItemMinThanArgumentValue`" for the functions `g1` and `g2` respectively.

This is unsurprising, because the program in Figure 1 is unidiomatic Python code. If one were to replace the function bodies for `g1` and `g2` with the more conventional:

```
def g1(l):
  def g11(n):
    return sorted(l)[n]
  return g11
```

and

```
def g2(l):
  return max(l)
```

respectively, then the system produces accurate names for each function: `get_sorted_values` for the top-level function f, and `get_nth_sorted_element` and `calculateMax` for `g1` and `g2` respectively.

The poor performance of baseline name suggestion techniques is not limited to LLMs: as we will see in Section IV, even Code2Vec [27], a state-of-the-art graph embedding-based name generation tool, produces poor quality names when applied to such unidiomatic code.

As we observe in our user study in Section V, nonsensical and misleading names massively inhibit program comprehension, and diminish the user's confidence in future synthesized code. In this context, the central problems that we consider in this paper are: (*a*) How do we provide additional information to the LLM in order to guide it towards better-chosen names? And (*b*) can we validate the names produced by the system before presenting them to the user? We will describe our solution to these problems in the rest of this section.

[1]For consistency with the user study, which was done with GPT-3.5, we will use this model throughout the paper. Informal experiments indicate that similar results would be obtained even with newer language models.

B. Prompt Expansion Using Subspecifications

Recall that our computational problem is to produce a name for each subroutine g that appears in the implementation f. The first part of our solution involves providing additional information to the language model about the role of g in the operation of f. For example, one might conceptually extend the function `g2` with instructions to log its execution:

```
def g2(x2):
  ...
  ans = list(filter(g21, x2))[0]
  print(f'g2({x2}) = {ans}') # Instrumentation
  return ans
```

Note that the original specification is in the form of input-output examples which can be mechanically evaluated to confirm that the implementation satisfies the spec. Upon testing the implementation with the logging code enabled, one finds that:

$$\texttt{g2}(\texttt{[1]}) = \texttt{1}, \tag{2}$$
$$\texttt{g2}(\texttt{[2, 1]}) = \texttt{2}, \tag{3}$$
$$\texttt{g2}(\texttt{[2, 7, 1]}) = \texttt{7}, \text{ and} \tag{4}$$
$$\texttt{g2}(\texttt{[9, 2, 7, 1]}) = \texttt{9}. \tag{5}$$

These observations immediately suggest that the function `g2` is computing the largest element of the list that it takes as input. While this does not provide conclusive proof, careful reading of the code confirms this hypothesis. In addition, one may conclude that *any* function `g2′` which satisfies Equations 2–5, regardless of whether or not it is otherwise semantically equivalent to `g2`, can be substituted into the original implementation of Figure 1 without affecting the fact that f satisfies the global specification, Equation 1.

This motivates us to extend the prompt supplied to the LLM with local input-output behavior of the function g being named. For example, for the function `g2`, we use the extended prompt shown in Figure 2.

With new information of this kind, the LLM is able to choose a more appropriate name for `g2`: `findLargestElement`. It also manages to recover the intent of the top-level function, f, for which it suggests the name `sortList`. In our experiments in Section IV, when applied to the list processing benchmarks solved by DreamCoder, providing logs of input-output behavior measurably improves the accuracy of names suggested by the LLM from 24% to 60% respectively.

Note II.1. The behavior of the function `g2`, as described by Equations 2–5, is closely related to the concept of subspecifications recently introduced by [36]. The major difference is that while subspecifications are necessary and sufficient conditions that characterize alternative implementations, monitoring input-output behavior merely provides sufficient conditions: there might conceivably be alternative implementations `g2′` that violate Equations 2–5 but which would nevertheless result in the global specification, Equation 1 being satisfied. Regardless, we will adopt their terminology, and refer to these input-output logs as the subspecifications of individual subroutines.

The primary technical difficulty in formalizing and obtaining these subspecs is the presence of higher-order functions. For example, naively instrumenting the function `g1` would produce outputs of the form:

```
g1([9, 2, 7, 1]) =<function g11 at 0x...>.
```

This output arises from the difficulty in serializing closures and higher-order functions. Our solution in Section III-A will involve a new specially designed interpreter to recover subspecifications for higher-order functions, yielding the result:

$$\texttt{g1}(\texttt{[9, 2, 7, 1]})(\texttt{1}) = \texttt{1}, \tag{6}$$
$$\texttt{g1}(\texttt{[9, 2, 7, 1]})(\texttt{2}) = \texttt{2}, \tag{7}$$
$$\texttt{g1}(\texttt{[9, 2, 7, 1]})(\texttt{3}) = \texttt{7}, \text{ and} \tag{8}$$
$$\texttt{g1}(\texttt{[9, 2, 7, 1]})(\texttt{4}) = \texttt{9}, \tag{9}$$

which immediately suggests that evaluating `g1(l)(n)` produces the `n`-th smallest element of the list `l`.

Unfortunately, even with this new information, in the run we consulted while writing this paper, the language model still suggested incorrect names for `g1` (the suggested name is "`genNextGreater-Value`") and the other subroutines in the implementation. Note however that responses from language models are inherently stochastic, so a subsequent run might not exactly reproduce these observations. In fact, our final implementation in NOMNOM makes productive use of this non-determinism.

> The implementation satisfies the specification. Choose a meaningful name for the function "`g2(x: List[int])-> int`":
> Specification:
>
> $$\texttt{g2}(\texttt{[1]}) = \texttt{1},$$
> $$\texttt{g2}(\texttt{[2, 1]}) = \texttt{2},$$
> $$\texttt{g2}(\texttt{[2, 7, 1]}) = \texttt{7}, \text{ and}$$
> $$\texttt{g2}(\texttt{[9, 2, 7, 1]}) = \texttt{9}.$$
>
> Implementation:
>
> ```
> def g2(x2):
>   def g21(x21):
>     def g22(x22):
>       return x21 < x22
>     return len(list(filter(g22, x2)))==0
>   return list(filter(g21, x2))[0]
> ```

Fig. 2: Example prompt for name generation when extended with local input-output subspecifications. We provide all prompt templates in Appendix B.

C. Algorithmic Sanity Checks

Our next insight is that when a function is appropriately named, that name can be used to substantially recover the original implementation. For example, recall that GPT-3.5 suggested the name `findLargestElement` for the function `g2`. Given this proposed function name and its type, we can request a

second language model to reproduce the corresponding function, to which it responds:

```
def findLargestElement(x: List[int]):
  maxValue = 0
  for val in x:
    if maxValue < val:
      maxValue = val
  return maxValue
```

Observe that this resynthesized implementation, `g2′ = findLargestElement`, is not semantically equivalent to the original function `g2`. In particular, it does not fail on empty inputs and it also assumes that the list does not contain any negative numbers. Despite these differences, observe that `g2′` continues to satisfy the same subspecification in Equations 2–5. It can therefore be substituted into the larger implementation of Figure 1 without affecting overall correctness, i.e., Equation 1. This suggests that `findLargestElement` is indeed an appropriate name for the subroutine `g2`.

Conversely, recall that with the extended prompt of Section II-B, the language model suggested the name `GenNextGreaterValue` for `g1`. Once again, we might ask the LLM to produce an alternative implementation of a function with this name, and type `List[int] -> Callable[[int], int]`. We provide a listing of its output in the supplementary material. Unsurprisingly, the implementation does not satisfy the subspecification corresponding to `g1`, i.e., Equations 6–9.

Although such checks do not guarantee the appropriateness of names, they at least confirm some degree of internal consistency. This gives us a mechanism to detect and filter out inappropriate names. In fact, in the run we consulted while writing this paper, this *algorithmic sanity check* provides support for the proposed names `findLargestElement` and `sortList` for the functions `g2` and f respectively. It also successfully refuted the spurious name suggestions for the remaining functions in the implementation.

Overall, in our experiments in Section IV, this further boosts the accuracy of the name suggestions from 60% to 82% respectively. On the other hand, notice that this technique is essentially a censor that filters out inappropriate names, whose use reduces the *response rate* from 97% to 42% respectively. One of the final optimizations in our system, NomNom, is a technique to exploit the non-determinism in LLM outputs and regenerate names upon failure of the sanity check. This manages to recover the drop in response rate from 42% to 72%, albeit with a slight decrease in accuracy from 82% to 79% respectively.

Note II.2. Given the extensive use of LLMs in our approach, the reader might wonder whether: (*a*) it might be possible to directly use the language model to synthesize code, and completely bypass the use of the underlying program synthesizer, and (*b*) whether the alternative implementation `g′` produced by the language model might somehow be more idiomatic and appropriate for presentation to the user. While this is certainly possible, this approach would sacrifice guarantees inherited from the underlying synthesizer, including that the implementation f satisfies the provided input-output examples.

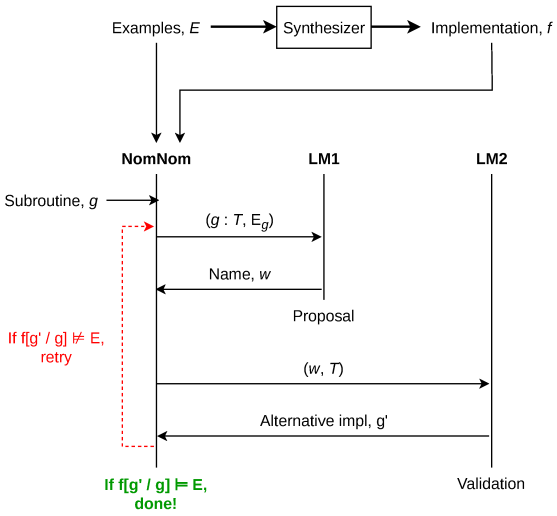


Fig. 3: Overall architecture of NomNom. We begin with a specification-implementation pair, (E, f), and a subroutine g of interest. The system alternates between querying a first LLM to obtain proposals w for g and validating w by resynthesizing an alternative implementation g' using a second language model.

In addition, many synthesis tasks are formulated in the context of a target DSL, and there is no guarantee that an implementation produced by a language model would follow the syntactic constraints of the target DSL.

We describe the overall architecture of NomNom in Figure 3. We describe its underlying algorithms in the next section.

III. Algorithmic Name Synthesis

We devote this section to describing our algorithm and some optimizations. The user starts by providing a set of input-output examples, $E = \{(i_1, o_1), (i_2, o_2), \ldots, (i_n, o_n)\}$. These examples may be drawn from integers, Boolean values, and lists of values. Upon successful synthesis, DreamCoder returns a program f which satisfies the specification E, i.e., for all $(i, o) \in E$, $f(i) = o$. We indicate this by writing $f \models E$. The implementations produced by DreamCoder are expressed as lambda-terms, examples of which may be found in the supplementary material. Given this specification-implementation pair, our system proposes meaningful names for each subroutine g appearing in f. We present the top-level procedure in Algorithm 1.

We begin by performing a best-effort analysis of the program, and associate each sub-expression t of the program with a type $t : T$. In addition, while presenting these programs to the user and to the language model, we freely alternate between their representations as lambda-terms and as programs expressed in a restricted subset of Python. For example, we present the lambda-term `(lambda (x)(cons (+ x 1)(cons (+ x 2) nil)))` to users as follows, inventing placeholder identifiers as needed:

```
def f(x):
  return [ x + 1, x + 2 ]
```

Algorithm 1 NOMNOM(E, f, g). Given a set of input-output examples $E = \{(i_1, o_1), (i_2, o_2), \ldots, (i_n, o_n)\}$, implementation $f \vDash E$, and a subroutine g of f, produces a name w for g.

1) Compute the local subspecification for g, $E_g = \{(i_{g1}, o_{g1}), (i_{g2}, o_{g2}), \ldots\}$.
2) Repeat until retries are exhausted:
   a) (PE.) Request a name w for g by supplying its type $g : T$ and the subspecification E_g and by using the prompt template from Figure 8b.
   b) Request an alternative implementation $g' : T$ of a function named w by using the prompt template from Figure 8c.
   c) (PE + SC.) Substitute the new implementation g' into f. If $f[g'/g] \vDash E$, then return the name w.
3) Report failure.

We now discuss the two principal elements of the algorithm, namely prompt expansion using subspecifications and the subsequent algorithmic sanity checks.

A. Prompt Expansion Using Subspecifications

As discussed in Section II-B, obtaining the local input-output examples E_g for prompt expansion is conceptually simple: one can place instrumentation code at appropriate points inside the function body, and log the inputs and outputs being sent into and produced by the function g currently being named. The hope is that the local input-output behavior provides clues to the overall purpose of g that is not apparent from its function body. However, this procedure is tricky because of the presence of higher-order functions. In particular, the function g might either itself take a function (closure) as input, or produce a closure as output, or possibly even both. Our solution is a custom interpreter that can print (serialize) closures. We provide additional details in Appendix A.

B. Algorithmic Sanity Checks

The names proposed by the LLM in response to the query in Step 2a are sometimes directly embedded in the original source code, or are presented with some other decoratory text, such as "*Name: «name»*". We have devised a set of simple extractor routines and regular expressions that detect these patterns and appropriately extract the proposed name. We hope to simplify this process by using structured prompting techniques in future versions of the system [37].

We then forward the proposed name w and the type T of the subroutine g being named to a second language model using the prompt template from Figure 8c. We interpret the response from the LLM as an alternative implementation g' of the original subroutine g. This step might fail either because the response from the LLM is not a syntactically well-formed program, or if it fails to have the desired type T, or if substituting it into the surrounding implementation compromises the overall correctness specification, $f[g'/g] \nvDash E$. In any of these cases, we reject the name w being proposed in response to the naming query in Step 2a. Note that we only generate names for top-level subroutines, so we do not have to consider the possibility of variable capture. This assumption greatly simplifies our implementation.

Finally, if $f[g'/g] \vDash E$, then we certify the name w as having passed the sanity check.

C. Optimizations

Finally, our implementation in NOMNOM includes two optimizations which increase the overall response rate of the system without losing accuracy.

a) Retries: It turns out that the algorithmic sanity checks of Section III-B are very effective in discovering inconsistencies between functions and their proposed names. Filtering names using this heuristic therefore massively improves the accuracy of the naming algorithm. Unfortunately, this accuracy improvement is accompanied by a corresponding drop in the number of queries successfully answered, as we will see in Section IV. The non-deterministic responses generated by language models provide an easy approach to mitigate this drop. When a proposed name fails the sanity check, we repeatedly retry (with a limit of 20 attempts) until the check succeeds, leading to the outermost loop in Algorithm 1.

b) Bottom-up name generation: Finally, there are certain functions which prove to be difficult to name even after multiple independent queries. One example is the following function f:

```
def f(x1):
  return a2(x1)(5)
def a2(x2):
  def a21(x21):
    def a22(x22):
      return x22>0
    return a22(countOccurrences(x2)(x21))
  return a21
```

Observe first that calling the function `a2(l)(n)` tests the output of `countOccurrences` to determine whether the value `n` occurs in the list `l`. It therefore follows that calling the top-level function $f(l)$ checks whether the provided list contains an occurrence of the number `5`.

The last optimization in NOMNOM facilitates this reasoning process by iteratively finding names for higher-level functions only after all lower-level functions, i.e., those reachable from it in the call graph have been successfully named.[2] In our experiments, this turns out to cause a slight increase in the response rate of the system, including for the function f above. This optimization has a flavor similar to emerging techniques for prompting language models such as scratchpads and chain-of-thought reasoning [38], [39].

IV. EXPERIMENTAL EVALUATION

Our implementation of NOMNOM uses `text-davinci-003` as our backend language model. We use the default language model settings for name generation, and only change

[2]Note that the concepts learned by DreamCoder naturally have a hierarchical structure in the form of a DAG.

`max_tokens` to 1,000 for the reverse code generation pass. Our evaluation focuses on the following research questions:

RQ1. How effective is our system in producing well-chosen names for subroutines?

RQ2. How frequently does the system produce suggestions for subroutine names?

RQ3. How many queries does the system require in order to propose these names?

a) Benchmarks: Our evaluation dataset started with 155 specifications involving list processing programs which were synthesized by DreamCoder.[3] Each of these specifications was associated with a name, indicating the user's intent, and a varying number of implementations (from 1 to 16). Upon manually inspecting these implementations, we discovered that 3 of them did not satisfy the stated user intent, and 8 implementations which we were unable to explain. We eliminated these programs, and chose the largest remaining implementation for each specification, which left us with a dataset consisting of 144 specification-implementation pairs and which consisted of a total of 344 subroutines. We manually provided reference names for each of these subroutines. The appropriateness of these names was subsequently validated by a visiting student researcher who was not among the authors of this paper.

b) Baselines: In addition to the baseline LLM and our algorithmic variants, we also evaluated the performance of Code2Vec [27]. Because Code2Vec works with Java code, we translated each of the benchmark programs into Java by hand. We will include these implementations in our artifact.

A. RQ1: Effectiveness of Explanations

In order to measure the effectiveness of NomNom in producing evocative function names, we ran five variants of our algorithm across the subroutines in our evaluation dataset. We then compared the algorithmically produced function names to our reference names and computed the Jaro similarity between the two [41], [42]. We declared the proposed name to be appropriate if this similarity measure exceeded 0.7. Finally, in order to estimate the variability of the overall procedure, we ran each algorithm five times for each subroutine. We present our observations in Figure 4.

Notice the consistent improvement in accuracy as we incorporate algorithmic improvements from a baseline score of 24% to the final value of 79%. One example of a successfully named function is the following:

```
def a2(x2):
  def a21(x21):
    def a22(x22):
      return x2[x22]
    return list(map(a22, range(x21)))
  return a21
```

Evaluating the function `a2(l)(n)` returns the first `n` elements of the list `l`. Only the last two algorithms consistently suggest

[3]File named `jobs/list_hard_test_ellisk_2019-02-15T11.43.28` from the DreamCoder artifact [40].

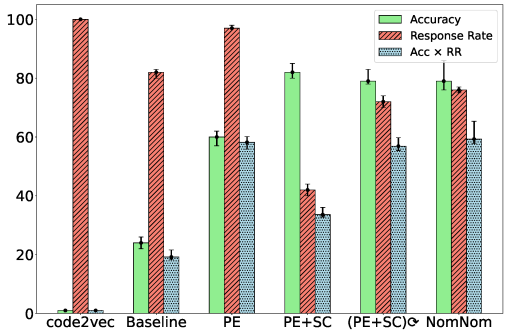


Fig. 4: Effectiveness of the algorithmic variants. PE indicates prompt expansion with local input-output subspecifications, PE + SC indicates the subsequent algorithmic sanity check, (PE + SC) ↻ indicates the version which repeatedly retries upon failure, and NomNom indicates our final system with bottom-up name generation. The bars represent the median of five independent executions.

names such as `getFirstNItems`, `selectFirstElements`, etc., while PE and PE + SC occasionally produce appropriate names. On the other hand, without any additional information, the baseline LLM suggests the misleading name, `transformList`.

Also, observe that Code2Vec generates poor quality names: this is because our benchmarks make heavy use of higher-order functions, resulting in unidiomatic Java code. In many cases, in the absence of an alternative, the system simply regurgitates the original placeholder function name, `f1`, `f2`, etc.

One concern with our evaluation methodology might involve the validity of the reference names. Of the 40 proposed function names polled in our user study, we only observed 3 names which we thought were appropriate (with sufficiently high Jaro similarity to the reference), but for which the average score of the users was "Neutral" or less. Conversely, we did observe situations where we marked a proposed name as inappropriate, even though users subsequently thought otherwise. One example is the following function `c2`:

```
def c2(x3):
  def c31(x31):
    def c32(x32):
      return x32==x31
    return len(list(filter(c32, x3)))
  return c31
```

Evaluating the function `c2(l)(k)` counts the number of occurrences of `k` in `l`. Our reference name was `countOccurrencesOfK`, while (PE + SC) ↻ suggested the name `countElementsMatchingValue`, which we rejected based on an insufficient Jaro similarity to the reference name.

B. RQ2: Response Rate of Tools

Although the algorithmic sanity check, PE + SC, significantly improves the accuracy of the overall algorithm, one concern is that suppressing responses from the LLM might lead to the overall system answering a smaller number of queries.

TABLE I: Confusion matrix from one run of PE + SC. We compare the suggested names to our reference names and declare a match when the Jaro similarity exceeds 0.7.

Reference	Filter Approves	Filter Rejects
Match	122 / 344	78 / 344
Mismatch	25 / 344	119 / 344

We therefore measured the response rate of the system, and we include this data in Figure 4.

First, observe that providing additional information to the language model, i.e., going from the baseline algorithm to one with prompt expansion, PE, modestly increases the overall response rate from 82% to 97% respectively. The real benefit of prompt expansion comes from the massive increase in the number of queries correctly answered, Acc × RR, from 19% to 58%.

Next, we observe that the greater accuracy of PE + SC is accompanied by a corresponding drop in response rate from 97% to 42% respectively. Indeed, it is not possible for a filtering pass to increase the total number of queries which are correctly answered, so that the product, Acc × RR, actually experiences a drop upon its application. We also provide the confusion matrix from one run of PE + SC in Table I. The overall F1 score of the filter turns out to be 0.70, so better filter designs is an important direction of future work.

Lastly, the figure also confirms the need for the final two algorithmic variants, (PE + SC) ↻ and NomNom: By giving the system multiple opportunities to produce an internally consistent response, they somewhat restore the response rate and provide modest increases in the product measure from 33% to 56% and 59% respectively.

Note IV.1. We repeated this experiment using the more recent `gpt-4-0125-preview` as the backend language model. The accuracy and the response rate of the baseline increased to 38% and 100% respectively, while the accuracy and the response rate of our final tool NomNom were 77% and 81% respectively.

C. RQ3: Number of LLM Queries Used

Finally, we measured the number of LLM queries needed by the different algorithmic variants to name each function. We list these statistics in Table II. Both the baseline approach and the variant with prompt expansion, PE, require just one LLM query to produce their response, while the version with algorithmic sanity checks enabled, PE + SC, needs two queries: the first to produce a name suggestion and the second to reverse-synthesize the subroutine body. On the other hand, the last two variants, with retries enabled, need to make additional queries when the first query either fails to elicit a response or receives a response which fails validation. Note that we report the median number of queries in Table II as the average is skewed by subroutines for which we hit the limit of 20 retries and eventually fail to produce a name.

While we did not explicitly track the time needed to name each subroutine or the cumulative cost of LLM queries, the statistics in Table II provide some guidance. Note that the time needed to name each subroutine is dominated by the response time from the OpenAI servers, and depends on numerous other factors such as load on the LLM implementation. In our experience, the most resource-intensive algorithms, (PE + SC) ↻ and NomNom, produce responses within 5–10 seconds for each subroutine.

TABLE II: Number of LLM queries needed by the algorithmic variants to name each subroutine. We report the median over five independent runs. The last 2 algorithms have higher query amounts because of failures listed in Section III.

Algorithm	Num Queries
Baseline	1
PE	1
PE + SC	2
(PE + SC) ↻	4
NomNom	4

V. User Study

To determine whether names help users in understanding the outputs of program synthesis tools, we conducted two user studies to answer the following questions:

RQ4. Do names help users in inferring the top-level purpose of each subroutine?

RQ5. Do names help users in understanding subroutines and the relationships between them?

RQ6. How do user preferences vary among the names produced by different algorithmic variants?

a) Participant selection: After IRB approval, we recruited 36 students who were familiar with Python from the engineering schools (Computer Science, Electrical Engineering, Mechanical Engineering, and Materials Science departments) of 7 prominent U.S. and Canadian universities. These participants had different levels of experience in programming and were a mix of undergraduate, Masters', and Ph.D. students. We posit that the variation in experience is representative of users of program synthesizers. We randomly divided the 36 participants into two groups with 18 participants for each user study.

A. Tasks and Study Structure

Before each user study, we had a short screening quiz asking participants to write a Python program that computes the sum of the elements in a list. Disregarding minor syntactic errors, all participants passed the screening quiz. We then showed participants a short video describing the tasks they needed to complete, and a brief introduction to aspects of the Python language that would be heavily used, including higher-order functions and some syntactic quirks. After the study was complete, we had a short discussion with each participant. During the discussion, participants gave their feedback about which aspects of the study they found easy or difficult, and their experience while answering questions. The study materials may be found in Appendices C and D.

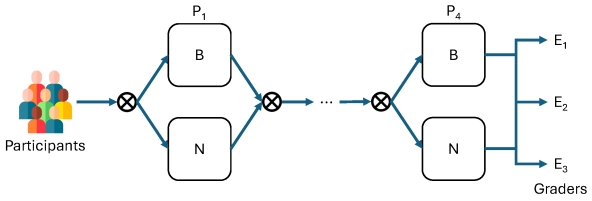


Fig. 5: Process followed for Study 1. 18 participants examined four programs with names either produced by the baseline LLM or by NOMNOM. We counterbalanced the study so each participant attempted two questions with names produced by the baseline, and the remaining two with names produced by our tool. Three domain experts independently graded their responses, and we report their average scores in Figure 6.

a) Study 1: Explanations: We chose four programs from the larger dataset of 144 implementations produced by DreamCoder with no mutual overlap in the subroutines used. In the first study, we asked participants to examine these programs and explain how they worked. We first asked participants to explain what each subroutine did, and then we asked them to walk us through the execution of the program on a specific input. We conducted the study in one of two randomly chosen conditions, with names suggested either by the baseline language model, or our final bottom-up name synthesizer, NOMNOM. We ensured that each participant attempted two tasks with names from the baseline approach, and the remaining two tasks with names from our system. In every case, the names were directly embedded inside the program.

For the first class of questions (i.e., what each subroutine did), we awarded responses with grades depending on whether it adequately captured the high-level goal of the function. For the second class of questions (i.e., walking us through an execution), we assessed whether participants accurately described how each function made use of the auxiliary functions that it called. Three domain experts independently provided these grades, and we present the average scores of participants for each question and condition in Figure 6.

b) Study 2: Preferences: The second study consisted of four tasks in which we presented users with a program and asked them to rate their preferences among different suggestions for function names on a five-point Likert scale ("*Inappropriate*", "*somewhat inappropriate*", "*neutral*", "*somewhat appropriate*", "*appropriate*"). We used the same programs as in the previous study. All participants undertook this study in the same condition, and had access to local subspecs for each subroutine. We measure the distribution of their responses in Figure 7.

B. RQ4: Understanding What Functions Do

We first measured the impact of the proposed function names on users' understanding of the top-level purpose of each function, and the effect of the different naming algorithms on this understanding. From Figure 6b, we observe that names suggested by our tool unambiguously help users in determining the purpose of each function.

Notice that for a majority of questions, all responses from participants looking at names from the baseline LLM were incorrect. We discovered that for the 12 subroutines in question, the baseline algorithm never suggested an appropriate name: four of its suggestions were nonsensical, and eight name suggestions were actually misleading.

By our estimate, all participants critically examined the programs shown to them. When faced with misleading names, they responded in a few different ways: one subset of participants chose to skip the question, another group of participants sensed a mismatch but chose to trust the stated names anyway, while the last subset disregarded the stated names upon discovering their inappropriateness and attempted to manually recover the function specification. In any case, apart from the program in Task 3, they were uniformly unable to discover the true purpose of the corresponding functions.

In contrast, all names produced by the final algorithm were appropriate, leading to massively higher accuracies when participants encountered questions in this condition.

C. RQ5: Understanding How They Work

Our next question involved determining the effect of names on users' understanding of how functions work. In particular, we asked users to walk us through an execution of the program, and focused on whether they were able to articulate the relationships among the various subroutines, i.e., why and how a particular subroutine called another.

These measurements correspond to the last question for each task in Figure 6b. Similar to RQ4, we concluded that participants have a much easier time understanding synthesized code when functions are appropriately named. During the post-study debrief, participants reported being surprised by "*unnatural*" code, and complained that the programs realize simple user intentions in complicated ways. In addition, even when they had well-chosen function names, effort was needed to confirm their understanding and frame their responses to the questions asked. When they were unable to definitively understand the code, some participants chose to guess functional relationships based on their names, while others opted to skip the question rather than make tentative predictions.

Anecdotally, we also found a cascading effect in users' understanding as they went higher up in the call graph towards the top-level functions: i.e., if they were unable to explain how a function f_2 used an auxiliary function f_3 which was called in its body, then they were often also unable to determine the working of a higher-level function f_1 which in turn called f_2.

D. RQ6: Distribution of User Preferences

In our final research question, we measured how user preferences varied among names produced by different naming algorithms. See Figure 7. From the figure, it is clear that the progressive algorithmic improvements that we discussed in this paper result in names that are well-liked by users. If we assign numerical values to these user preferences on a 0 ("*Inappropriate*")–1 ("*Appropriate*") scale, then the average score of names produced by the baseline LLM is 0.11,

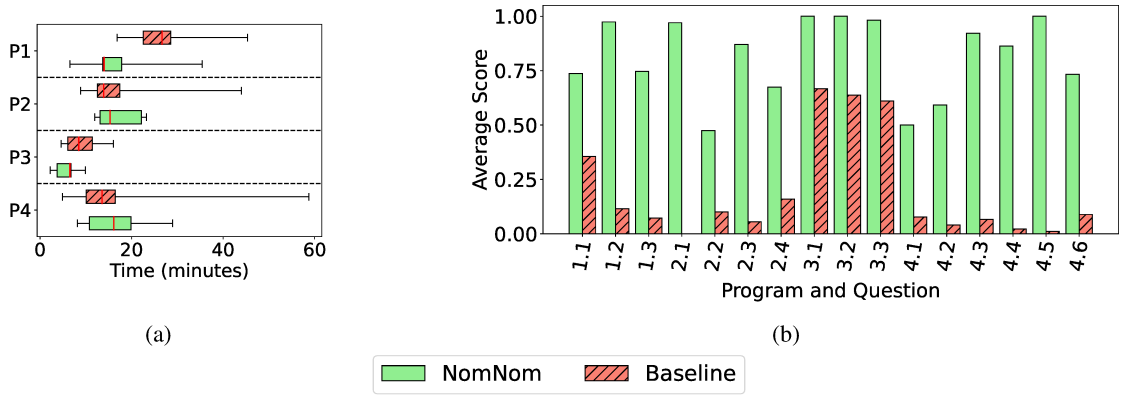


Fig. 6: Time needed and accuracy of responses when explaining how implementations worked (Study 1). Each bar is the average score awarded by three domain experts, with pairwise Pearson correlation coefficients of 0.85, 0.80, and 0.83 respectively.

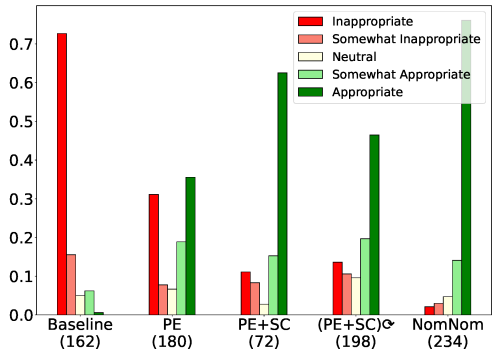


Fig. 7: Distribution of participant preferences among names suggested by different algorithms (Study 2). The numbers in parentheses indicate the total number of user responses collected for the corresponding naming algorithm.

while the average score of names produced by NomNom is 0.89. Furthermore, response rates show a similar trend as in Section IV, with PE + SC producing the fewest suggestions.

Anecdotally, participants who undertook this study (Study 2) found it easier to indicate their preferences, as compared to participants who had to provide more detailed accounts of how the implementations worked (Study 1). Recall that we included the local input-output subspecifications as part of this study: most participants made their judgments by simply checking for compatibility between the proposed names and the subspecs.

VI. Limitations and Discussion

We now discuss some limitations of our approach and the evaluation methodology employed in this paper.

The first concern is whether function names are sufficient to help programmers understand the synthesized implementation. How programmers choose identifier names [43], [44], and their effect on program comprehension [45] has been the subject of extensive research. Despite some research indicating otherwise [46], there is broad agreement that good variable names are important [47]. This is consistent with our observations in Section V, where programmers are able to better understand code with appropriately chosen function names.

We might instead have asked the language model to produce explanations in the form of longer free-form comments: Note that our thesis in Section III-B is that well-chosen function names can be used to recover semantically equivalent implementations, and can thereby be subject to experimental falsification. The experimental validation of free-form comments, whether produced by a language model or a human programmer, is an important challenge for future work.

An interesting related question in this context is how the system would respond to project-specific naming conventions, requiring, for example, the use of Hungarian notation or camelcase identifiers. An easy solution is for the programmer to adapt the function name as they see fit. A second solution might be to extend the LM1 prompt to ensure that function names follow the necessary conventions. A final option is to also validate the appropriateness of suggested identifiers as part of the sanity check procedure.

The reader may also be concerned about our choice of DreamCoder as the background synthesizer. First, we note that DreamCoder is an example of a larger family of library learning tools [19], [33], [35], [48] which aim to learn libraries of reusable components within a domain specific language. Furthermore, name generation using subspecifications and sanity checks is more broadly applicable to other program synthesizers. For example, consider the following program generated by λ^2 [30]:

```
def g1(x1):
  def g11(x11):
    def g12(x121, x122):
      return x121 or (x122 < x11)
    return functools.reduce(g12, x1, False)
  return list(filter(g11, x1))

def f(x1):
  return list(map(g1, x1))
```

Simulating our technique by hand, we derived the names `removeInnerSmallest` and `removeSmallestElement` for the functions `f` and `g1` respectively. More generally, systems

such as NomNom might be useful both to obtain more idiomatic code from program synthesizers, and as interactive mechanisms to help users understand code. Conversely, it would be an interesting question to see whether ideas similar to ours can be used to more reliably synthesize code using LLMs.

Another concern involves the brittleness of LLM outputs in response to minor changes in the provided prompt. While this cannot be completely mitigated, we hope that the full text of prompts provided in the supplementary material will at least partially address issues with reproducibility.

One might also object to human-written reference names being used for evaluation in Section IV. To mitigate this concern, we had our reference names cross-verified by another student programmer not among the authors of this paper.

A final concern involves potential biases in our choice of participants for the user study in Section V. Are these participants representative of *actual* users of program synthesis tools? While we attempted to mitigate this concern by drawing broadly from graduate and undergraduate students across engineering schools of various prominent American and Canadian universities, conducting a larger study with working programmers is an important direction of future work.

VII. Related Work

a) Comprehension of Synthesized Programs: While program synthesizers are meant to realize logical correctness specifications, their output is rarely easy to understand. This obscurity partly stems from the use of procedurally generated identifier names (`v1`, `v2`, `f1`, `f2`, etc.) in synthesized code.

Human programmers use different practices to make code comprehensible, such as by writing detailed comments [49], using meaningful names and identifiers [50], or maintaining documentation of their rationale. Of these approaches, the use of meaningful names has been found to make a significant contribution to improve the comprehensibility [47]. This is intuitive, as 70% of source code consists of identifiers [45]. Thus, the lack of meaningful names in synthesized code inhibits understanding. Although the automatic generation of meaningful function and variable names is a well-studied problem in software engineering: see [51] for a survey, and Code2Vec [27] and JSNice [28] for prominent examples. Still, the unintuitive nature of automatically generated programs makes it challenging to apply existing techniques.

Researchers have investigated the nature of comprehensible code [52], and studied techniques to help improve comprehension like program debugging [53], [54], slicing [55], [56], automatic summarization [57], and user-guided program synthesis [58]. However, none of these techniques have been used to generate names for synthesized programs. [36] introduced subspecs to allow programmers to reason about individual parts of synthesized code. Our paper investigates how subspecs can be used to help LLMs produce meaningful identifier names.

b) Large Language Models in Program Synthesis: LLMs have been shown to be surprisingly capable of generating code from natural language specifications of programmer intent [25]. Such LLMs also have the potential to improve the explainability of code by augmenting it with natural language explanations. However, these large language models do not understand program semantics, and offer no guarantees about quality and accuracy of the suggested code or explanations.

To provide guarantees with LLMs, researchers have suggested using LLMs as a complementary approach to formal methods which can guarantee accuracy and adherence to specifications. For example, Jigsaw [59] is a program synthesis tool that augments LLMs with post-processing steps based on program analysis and synthesis techniques, NLX [60] marries pre-trained natural language models and component-based program synthesis for multi-modal program inference. In this paper, we follow this approach of combining program synthesis and LLMs to augment synthesized programs with explanatory names that help users understand the intent of the code.

c) Prompting: The effective use of LLMs depends on carefully chosen prompts: Researchers have investigated various prompting techniques like LLM programming [37] which is a combination of text prompting and scripting, chain-of-thought prompting [39] which uses a series of intermediate reasoning steps to perform complex reasoning, and probabilistic inference paradigm [61] which probabilistically reasons over sets of objects using LLMs. While researchers have investigated these approaches to improve the accuracy of language models, it is not evident how to use these techniques to justify the appropriateness of names for program elements.

In this paper, we discuss two novel prompting approaches to improve the accuracy of explanatory names generated by LLMs for synthesized programs. First, we expand the prompt using subspecifications by recovering input-output data from the synthesis algorithm. Second, we conduct sanity check of the algorithm by using another language model to validate the proposed explanations.

VIII. Conclusion

In this paper, we showed how to use a large language model (LLM) to annotate automatically synthesized code with meaningful names. Our procedure principally relies on a combination of two relatively simple techniques: *first*, by including additional information about the implementation as part of the prompt, and *next*, by validating the proposed names with an algorithmic sanity check. Both experiments and a user study show the effectiveness of our technique in producing well-chosen function names. Our research contributes to the emerging body of work on combining language models with formal reasoning techniques. In future, we hope to extend these ideas to automatically produce free-form comments, and also potentially techniques that confirm the validity of comments, identifiers and other natural language artifacts in code.

The artifact supporting the claims in this paper may be downloaded from https://doi.org/10.5281/zenodo.12682854.

Acknowledgments

We thank all the participants in our user study and the anonymous reviewers for immeasurably improving this paper. The research described in this paper was supported by the NSF under grants CCF #2146518, #2124431, and #2107261.

REFERENCES

[1] S. Gulwani, A. Polozov, and R. Singh, *Program Synthesis*. NOW, August 2017, vol. 4. [Online]. Available: https://www.microsoft.com/en-us/research/publication/program-synthesis/

[2] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, p. 84–93, nov 2018. [Online]. Available: https://doi.org/10.1145/3208071

[3] A. Udupa, A. Raghavan, J. Deshmukh, S. Mador-Haim, M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2013, p. 287–296.

[4] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 10205, 2017, pp. 319–336.

[5] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. ACM, 2018, p. 420–435.

[6] T. Lau, P. Domingos, and D. Weld, "Version space algebra and its application to programming by demonstration," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML. Morgan Kaufmann Publishers Inc., 2000, pp. 527—534.

[7] P.-M. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2015, pp. 619—630.

[8] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova, "Program synthesis by type-guided abstraction refinement," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, Dec. 2019.

[9] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. ACM, 2018, p. 436–449.

[10] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS. ACM, 2013, p. 305–316.

[11] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, "Chlorophyll: Synthesis-aided compiler for low-power spatial architectures," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2014, pp. 396—407.

[12] J. McClurg, H. Hojjat, and P. Černý, "Synchronization synthesis for network programs," in *Computer Aided Verification*. Springer, 2017, pp. 301–321.

[13] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL. ACM, 2011, p. 317–330.

[14] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, p. 816–827, Jun. 2016.

[15] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2014, pp. 542—553.

[16] L. Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantiative objectives," in *27th International Conference on Computer Aided Verification*, ser. CAV, July 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*, ser. ICSE. IEEE Press, 2013, p. 772–781.

[18] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2013, p. 15–26.

[19] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. Tenenbaum, "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*, ser. PLDI 2021. ACM, 2021, p. 835–850.

[20] R. Ji, Y. Sun, Y. Xiong, and Z. Hu, "Guiding dynamic programing via structural probability for accelerating programming by example," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, Nov. 2020.

[21] L. Laich, P. Bielik, and M. Vechev, "Guiding program synthesis by learning to generate examples," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BJl07ySKvS

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[23] OpenAI, "Gpt-4 technical report," 2023.

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[25] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[26] "Github copilot," https://copilot.github.com/, 2021.

[27] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290353

[28] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. ACM, 2015, p. 111–124.

[29] M. Allamanis, E. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[30] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 229–239. [Online]. Available: https://doi.org/10.1145/2737924.2737977

[31] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 522–538. [Online]. Available: https://doi.org/10.1145/2908080.2908093

[32] A. Miltner, A. T. Nuñez, A. Brendel, S. Chaudhuri, and I. Dillig, "Bottom-up synthesis of recursive functional programs using angelic execution," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498682

[33] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, 2023. [Online]. Available: https://doi.org/10.1145/3571207

[34] A. Pal, B. Saiki, R. Tjoa, C. Richey, A. Zhu, O. Flatt, M. Willsey, Z. Tatlock, and C. Nandi, "Equality saturation theory exploration á la carte," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 2023. [Online]. Available: https://doi.org/10.1145/3622834

[35] M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, and A. Solar-Lezama, "Top-down synthesis for library learning,"

Proc. ACM Program. Lang., vol. 7, no. POPL, jan 2023. [Online]. Available: https://doi.org/10.1145/3571234

[36] A. Nazari, Y. Huang, R. Samanta, A. Radhakrishna, and M. Raghothaman, "Explainable program synthesis by localizing specifications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622874

[37] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," vol. 7, no. PLDI, 2023.

[38] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena, "Show your work: Scratchpads for intermediate computation with language models," 2021.

[39] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *CoRR*, vol. abs/2201.11903, 2022.

[40] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. Tenenbaum, "Dreamcoder software and data," 2021. [Online]. Available: https://doi.org/10.1145/3410302

[41] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1989.10478785

[42] W. Cohen, P. Ravikumar, and S. Fienberg, "A comparison of string metrics for matching names and records," in *Kdd workshop on data cleaning and object consolidation*, vol. 3, 2003, pp. 73–78.

[43] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer, "How developers choose names," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 37–52, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.2976920

[44] A. Swidan, A. Serebrenik, and F. Hermans, "How do scratch programmers name variables and procedures?" in *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*. IEEE Computer Society, 2017, pp. 51–60. [Online]. Available: https://doi.org/10.1109/SCAM.2017.12

[45] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, pp. 261–282, 2006.

[46] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. IEEE Press, 2017, p. 45–54. [Online]. Available: https://doi.org/10.1109/ICPC.2017.18

[47] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension: An empirical study," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 55–65.

[48] C. Wong, K. M. Ellis, J. Tenenbaum, and J. Andreas, "Leveraging language to learn program abstractions and search heuristics," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 11 193–11 204. [Online]. Available: https://proceedings.mlr.press/v139/wong21a.html

[49] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, p. 215–223.

[50] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–40. [Online]. Available: https://doi.org/10.1145/3196321.3196332

[51] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2019, pp. 602–614.

[52] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 181–191.

[53] R. Caballero, A. Riesco, and J. Silva, "A survey of algorithmic debugging," *ACM Computing Surveys*, vol. 50, pp. 1–35, 08 2017.

[54] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE. Springer, 1999, pp. 253–267.

[55] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE. IEEE Press, 1981, pp. 439–449.

[56] A. Ko and B. Myers, "Designing the Whyline: A debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI. ACM, 2004, pp. 151–158.

[57] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," 2019.

[58] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. Glassman, "Interpretable program synthesis," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, 2021.

[59] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE. ACM, 2022, pp. 1219–1231.

[60] K. Rahmani, M. Raza, S. Gulwani, V. Le, D. Morris, A. Radhakrishna, G. Soares, and A. Tiwari, "Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, oct 2021.

[61] B. Ozturkler, N. Malkin, Z. Wang, and N. Jojic, "Thinksum: Probabilistic reasoning over sets using large language models," 2023.

[62] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae*, vol. 75, no. 5, pp. 381–392, 1972.

[63] M. Shulman, "You could have invented de Bruijn indices," https://golem.ph.utexas.edu/category/2021/08/you_could_have_invented_de_bru.html, 2021.

APPENDIX

A. Algorithmic Name Synthesis

We now describe our custom interpreter that can compute subspecifications in the presence of higher-order functions. First, our interpreter includes a mechanism to print (serialize) closures. Thus, for example, the subspecification of `g1` in the following program:

```
def g1(h):
  return h(3)
def f1(x):
  return g1(lambda y: x + y)
```

given the global input-output example $f_1(2) = 5$ is given by:

$$\texttt{g1}(\texttt{lambda y: 2 + y}) = 5.$$

Notice that this constraint is satisfied by the original subroutine `g1`, but is also satisfied by other implementations, including by the constant-valued function $\texttt{g1}' = \texttt{lambda y: 5}$. Furthermore, it is safe to use any such new implementation $\texttt{g1}'$ instead of the previous implementation `g1`.

Next, we consider the case when subroutines themselves return closures. Consider, for example, the subroutine `g2` in the following program:

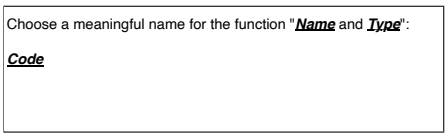
```
def g2(x):
  def h(y): return x + y
  return h
def f2(x):
  return g2(x)(3)
```

With the global input-output example $f_2(2) = 5$, the subroutine `g2` is invoked with the input `x = 2`. If we instrument the return value of `g2` as before, we would observe that $\texttt{g2}(2) = \texttt{lambda y: 2 + y}$. Notice that, given an alternative implementation $\texttt{g2}'$, it is conceptually difficult to compare closures for equality.

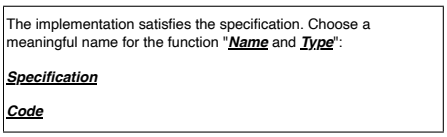
We instead use the following procedure. Say we wish to produce the subspec for a function `g` which itself produces a closure as output. In this case, given the original argument $\texttt{x}_1$, we wrap the closure $\texttt{g}(\texttt{x}_1)$ in a monitor object $m_1 = ([\texttt{x}_1], \texttt{g}(\texttt{x}_1))$. We incorporate successive arguments to m_1, say $\texttt{x}_2$, $\texttt{x}_3$, ..., $\texttt{x}_k$, resulting in the monitor object $m_k = ([\texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_k], \texttt{g}(\texttt{x}_1)(\texttt{x}_2)(\cdots)(\texttt{x}_k))$. When the closure finally reduces to a ground value, the monitor $m = (l, v)$ prints the sequence of function arguments l and the finally produced ground value v, which are then included as part of the subspecification E_g.

In the example above, recall that the global specification was $f_2(2) = 5$. Querying the subspecification for `g2` initially results in the monitor object $m_1 = ([2], \texttt{lambda y: 2 + y})$. Further evaluation using this monitor object produces the final subspec $\texttt{g2}(2)(3) = 5$. We can show that:

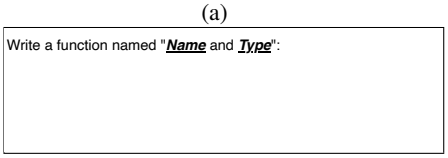
Lemma A.1. *Let E be a set of input-output examples, and $f \vDash E$ be a conformant implementation. Let g be a subroutine in f, with local input-output subspecification E_g. Pick a function g' such that $g' \vDash E_g$. Then $f[g'/g] \vDash E$.*



(a)



(b)



(c)

Fig. 8: Prompt templates utilized in the generation of names and code using the LLMs. Figure 8a is utilized to generate names for the baseline, while Figure 8b is employed to generate names for our algorithms. Additionally, Figure 8c is employed to generate programs for our filtering technique.

```
(lambda
  (map (lambda
         #(lambda
            (lambda
              #(lambda
                 (car (filter $0 (lambda
                                   (empty? (filter $1 (lambda
                                                        (gt? $0 $1)))))))))
                 (filter $1 (lambda
                              (gt? $1 (length (filter $2 (lambda
                                                           (gt? $1 $0)))))))))
       $1 (+ 1 $0)))
       (range (length $0)))
```

Fig. 9: Sorting a list of numbers. The corresponding Python program is in Figure 13.

B. Prompt Templates

Figure 8 displays the prompt templates utilized in the generation of names and code using the LLMs.

C. Implementation Listings

We list the lambda-expressions emitted by DreamCoder in Figures 9–12 and our corresponding transliterated Python programs for all examples and user study tasks in Figures 13–16. Note that the lambda-expressions use de Bruijn indices [62], [63], thereby eliminating variable names.

```
(lambda (#(lambda (lambda ((lambda (gt? $0 0))
      (#(lambda (lambda (length (#(lambda (lambda (#(lambda (lambda
          (fold $1 empty
                (lambda (lambda (fold ($2 $1 range) $0
                                      (lambda (lambda (cons $3 $2))))))))
          $1 (lambda (lambda (if ($2 $1) $3 empty)))))) $1
                (lambda (eq? $0 $1)))))) $0 $1))))
  (+ (+ (+ 1 (+ 1 1)) 1) 1)
  $0)
```

Fig. 10: Checking if a list has 5. The corresponding Python program is in Figure 14.

```
(lambda (cdr (
    #(lambda (lambda (map (lambda (index $0 $2)) (range $0))))
         (map (lambda $0) $0) (+ 1 (+ 1 1)))))
```

Fig. 11: Getting second and third elements of a list. The corresponding Python program is in Figure 15.

```
(lambda (fold $0 (#(map (lambda (mod (+ $0 1) (+ 1 (+ 1 1))))) empty)
    (lambda (lambda (#(lambda (lambda (fold $1 (cons $0 empty)
        (lambda (lambda (cons $1 $0))))))) $0 $1))))
```

Fig. 12: Reversing a list. The corresponding Python program is in Figure 16.

```
def a3(x3):
    def a31(x31):
        def a32(x32):
            return x31 < x32
        return len(list(filter(a32, x3)))==0
    return list(filter(a31, x3))[0]

def a2(x2):
    def a21(x21):
        def a22(x22):
            def a23(x23):
                return x22>x23
            return x21>len(list(filter(a23, x2)))
        return a3(list(filter(a22, x2)))
    return a21

def a1(x1):
    def a11(x11):
        return a2(x1)(x11+1)
    return list(map(a11, range(len(x1))))
```

Fig. 13: Sorting a list of numbers.

D. User Study Tasks

Figure III includes all names generated by all algorithms used in two user studies.

TABLE III: The generated names used in two user studies. In the first user study, we only used names generated by the baseline and NOMNOM. In the second user study, we asked participants to rate the names generated by the baseline, PE, PE + SC, (PE + SC) ↻, and NOMNOM. We omitted the names produced by Code2Vec because they were uniformly uninformative.

Function	Code2Vec	Baseline	PE	PE+SC	(PE+SC) ↻	NomNom
T1.a3	a3	foldFuncComposer				appendListFunc
T1.a2	a2	addModuloThree	getScaledX21Value	modulo_plus_one	modulo_plus_one	addOneModThree
T1.a1	a1		reverseOrderList	reverseList	reverseMapping	reverseList
T2.a3	a3	getLowestUnusedValue	getMaxElement		getMaxElement	getLargestElement
T2.a2	a2	getHighestValueInList	getNthElementInList			getOrderedElement
T2.a1	a1	findMaxElementIndex	findMaxElementIndex		sortList	sortList
T3.a2	a2	transformList			getFirstXElements	getFirstNItems
T3.a1	a1				extractNextTwoElements	getSecondAndThirdElements
T4.a5	a5		filterFives		filterAppendFunc	filterFunc
T4.a4	a4		filterIntListByPredicate	filterListByPredicate	filterList	filterListByPredicate
T4.a3	a3	summationFunc	countOccurrences		countElementsMatchingValue	countOccurrences
T4.a2	a2	filterPositiveNumbers	hasElementsGreaterThan	containsNumber	containsNum	hasAtLeastOneOccurrence
T4.a1	a1	isOddIntList	isEqual5s		checkForFiveInSequences	hasFiveInList

```
import functools

def a5(x5):
    def a51(x51):
        def a52(x52, x53):
            def a54(x54, x55):
                return [x53]+x52
            return functools.reduce(a54, x51(x53)(range)[::-1], x52)
        return functools.reduce(a52, x5[::-1], [])
    return a51

def a4(x4):
    def a41(x41):
        def a42(x42):
            def a43(x43):
                if x41(x42):
                    return x4
                else:
                    return []
            return a43
        return a5(x4)(a42)
    return a41

def a3(x3):
    def a31(x31):
        def a32(x32):
            return x32==x31
        return len(a4(x3)(a32))
    return a31

def a2(x2):
    def a21(x21):
        def a22(x22):
            return x22>0
        return a22(a3(x2)(x21))
    return a21

def a1(x1):
    return a2(x1)(((1+(1+1))+1)+1)
```

Fig. 14: Checking if a list has 5.

```
def a2(x2):
    def a21(x21):
        def a22(x22):
            return x2[x22]
        return list(map(a22, range(x21)))
    return a21

def a1(x1):
    def a11(x11):
        return x11
    return a2(a11(x1))(1+(1+1))[1:]
```

Fig. 15: Getting second and third elements of a list.

```
import functools

def a3(x3):
    def a31(x31):
        def a32(x32, x33):
            return [x33] + x32
        return functools.reduce(a32, x3[::-1], [x31])
    return a31

def a2(x2):
    return (x2+1)%(1+(1+1))

def a1(x1):
    def a11(x11, x12):
        return a3(x11)(x12)
    return functools.reduce(a11, x1[::-1], list(map(a2, [])))
```

Fig. 16: Reversing a list.

```
  def genNextGreaterValue(x):
    def f(i: int):
        greater = []
        for e in x:
            if e > i:
                greater.append(e)
        return min(greater)
    return f
```

Fig. 17: Generated code by LLM for the generated name `genNextGreaterValue`.