Program Synthesis using Large Language Models

By

Saksham Sahai Srivastava

Advised By

Prof. Ashutosh Trivedi

&

Prof. Gowtham Kaki

Date: Jul 11, 2023

Department of Computer Science

University of Colorado Boulder

Contents

1 Abstract

Program Synthesis is the task to construct a program that provably satisfies a given high-level formal specification. In this project, we explore the application of large language models(LLMs) especially chatGPT, for synthesizing functional programs in Ocaml. We leverage LLMs for this task due to their ability to allow for seamless integration of high-level specifications into the program synthesis process.

We utilize a comprehensive range of benchmarks for lists, trees and nested data structures sourced from prominent program synthesis papers to assess the effectiveness of our approach. Through this evaluation, we aim to gain insights into the capabilities of chatGPT in synthesizing programs as compared to traditional program synthesis techniques.

2 Introduction

Synthesis of correct and effective programs has the potential of revolutionizing software development because it enables developers to shift their focus more towards problem-solving rather than manually things down. Albarghouthi[1] proposed Escher which is a program synthesis algorithm which basically interacted with the user via input-output examples to generate recursive programs which matched the intended behavior. But this technique had a drawback that it was primarily designed to synthesize recursive programs. However, not all programs are best expressed or implemented using recursion. Then, Leon proposed by Knuess[2] is an automated system that is able to generate software that manipulates 'unbounded' data types, like numbers and data structures without a fixed size. It can also take specifications and turn them directly into code that is verified to be correct for all possible inputs. But this technique holds a drawback of scalability, which means it struggles in generating large and complex pieces of code. Not only that it focuses more on producing code that meets functional requirements rather than maintaining a specific structure or coding style.

Later, Lambda Square(λ^2) introduced by Feser[3] presented a method for synthesizing functional programs over recursive data structures, using input-output examples. This technique uniquely combined inductive generalization, deduction, and best-first enumerative search to guarantee the simplest possible program fitting the examples. This technique was more scalable as compared to Leoan[2]. Then, Polikarpova[4] introduced a program synthesis method using polymorphic refinement types which offered automatic verification of complex programs and enabled a significant reduction in search space size. At its core this algorithm for refinement type checking supported the breakdown of specifications for independent components. While refinement types are powerful, they may not be able to express all specifications or constraints, especially when these are complex or involve aspects not covered by decidable logics. Lubin[5] proposed Smyth which outperformed Leon[2] and Synquid[4] which were the existing state-of-the-art synthesis tools. It is a system for program sketching in a typed functional language that utilized live bidirectional evaluation to propagate input-output examples through partially completed program sketches. This innovative method enabled the synthesis of recursive functions without requiring trace-complete sets of examples, and allowed for the specification and solving of interdependent synthesis goals. But it had a drawback of requiring significant computational resource to propagate examples backward through partially evaluated sketches, especially for larger or more complex programs. Recently, Miltner[6] came up

with a synthesis technique called Burst where they present a bottom-up synthesis technique for functional recursive programs, overcoming prior challenges by incorporating angelic semantics and a process of specification strengthening. It uses an angelic synthesis technique based on finite tree automata, enabling efficient handling of increasingly complex specifications.

We are using large language models for our task of program synthesis because it can extend the capability of Escher to generate recursive programs via user-provided input-output examples. Another compelling aspect is the handling of 'unbounded' data types by systems like Leon, which also offers the creation of verified code across all possible inputs. LLMs, given their inherent expertise in understanding and generating complex natural language, could provide a scalable solution to the challenges faced by existing methods in generating large and complex pieces of code. The method employed by Lambda Square for synthesizing functional programs over recursive data structures showcases a similar potential for augmentation through LLMs. Synquid's methodology offers automatic verification of complex programs and significant reduction of the search space, hinting at another domain where LLMs are likely to excel.

3 Experiments

The experiment was conducted in three parts. First, when the problem specification was given as input to the large language model. Second, when the input-output examples were given as input but the inputs were tainted with keywords which the LLMs might be familiar with. Lastly, input-output examples were fed to the LLM but the data structures were customly defined so that the LLM cannot exploit it's recall ability. For simplicity, the programs are synthesized in **Ocaml** functional programming language. This choice ensured consistency and enabled comparison with pre-existing methods and techniques.

3.1 Experiment - 1

In this part of the experiment, for example, we want to synthesize a *concatenate_list_of_lists* function. So, a natural language prompt was given as an input to the LLM model which completely describes the required specification such as "*Write an Ocaml program which takes a nested list of lists as an input and converts it to a single flat list, where all elements are arranged sequentially.*"

The output obtained was as follows:

```
let rec concatenate_lists lst =
  match lst with
  | [] -> []
  | hd :: tl -> hd @ concatenate_lists tl

let () =
  let test_lst = [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]] in
  let result = concatenate_lists test_lst in
  List.iter (Printf.printf "%d ") result;
  print_newline ()
```

3.2 Experiment - 2

In this part of the experiment, we completely shift our focus to leveraging input-output examples for the objective of synthesizing programs. Let's say we want to generate a program that counts all the nodes of the tree. So, a natural language prompt was given as *"Write the Ocaml code which satisfies the logical specification for the given input-output examples. The examples are as follows:"*

Example - 1:

$$\begin{aligned}&\text{Node}\,(4,\\&\quad\text{Node}\,(2,\\&\qquad\text{Node}\,(1,\text{Empty},\text{Empty})\,,\\&\qquad\text{Node}\,(3,\text{Empty},\text{Empty})\\&\quad)\,,\\&\quad\text{Node}\,(5,\\&\qquad\text{Node}\,(10,\text{Empty},\text{Empty})\,,\text{Empty}\\&\quad)\\&)\rightarrow 6\end{aligned}$$

Example - 2:

$$\begin{aligned}&\text{Node}\,(6,\\&\quad\text{Node}\,(8,\\&\qquad\text{Node}\,(2,\\&\qquad\quad\text{Node}\,(12,\text{Empty},\text{Empty})\,,\text{Empty}\\&\qquad)\,,\\&\qquad\text{Node}\,(11,\text{Empty},\text{Empty})\\&\quad)\,,\\&\quad\text{Node}\,(10,\\&\qquad\text{Node}\,(3,\text{Empty},\text{Empty})\,,\text{Empty}\\&\quad)\\&)\rightarrow 7\end{aligned}$$

The output obtained was as follows:

```
let rec count_nodes = function
  | Empty -> 0
  | Node (_, left, right) -> 1 + count_nodes left + count_nodes right;;

let () =
  let t1 = Node (4,
                 Node (2,
                     Node (1, Empty, Empty),
                     Node (3, Empty, Empty)),
                Node (5, Node (10, Empty, Empty), Empty)) in
```

```
let t2 = Node (6,
               Node (8,
                   Node (2, Node (12, Empty, Empty), Empty),
                   Node (11, Empty, Empty)),
              Node (10, Node (3, Empty, Empty), Empty)) in

print_endline (string_of_int (count_nodes t1));  (* prints "6" *)
print_endline (string_of_int (count_nodes t2));  (* prints "7" *)
```

In this experiment, the LLM model might have got hints from the keywords Node, Empty etc. to exploit it's recall ability and thus spitting out the Ocaml code for node count.

3.3 Experiment - 3

In this part of experiment, we specifically tailored and designed custom data structures for input-output examples so as to challenge the LLM's recall ability and prevent it from relying solely on previously encountered patterns or pre-learned information. This helps in accessing LLM's adaptability and ability to synthesize programs based on completely new scenarios.

We define data structures in the following fashion:

List:

```
type 'a dd = Zyx | Xyz of 'a * 'a dd
```

Binary Tree:

```
type 'a p =
  | m1
  | m2 of 'a * 'a z

and 'a q =
  | n1
  | n2 of 'a * 'a z

and 'a z =
  | c of 'a p * 'a q
```

Binary Tree of Lists:

```
type 'a i =
  | m1
  | m2 of 'a dd * 'a k

and 'a j  =
  | n1
```

```
  | n2 of 'a dd * 'a k

and 'a k =
  | c of 'a i * 'a j
```

But this information of customly defining the data structure was not shared with chatGPT, instead we directlt gave input-ouput examples to it. Now, let's say we want to flatten a binary tree of lists. So, the natural language prompt will be given as follows: " *Write the Ocaml code which satisfies the logical specification for the given input-output examples. The examples are as follows:*"

Example - 1:

$$
\begin{array}{l}
c\,(\\
\quad m2\,(Xyz(2, Xyz(1, Zyx)), \\
\quad\quad c\,(\\
\quad\quad\quad m2\,(Xyz(3, Zyx), \\
\quad\quad\quad\quad c\,(\\
\quad\quad\quad\quad\quad m2\,(Zyx, m1, m1)\,, \\
\quad\quad\quad\quad\quad m2\,(Xyz(11, Xyz(8, Zyx)), m1, m1) \\
\quad\quad\quad\quad)\, \\
\quad\quad\quad)\,, \\
\quad\quad\quad m2\,(Zyx, \\
\quad\quad\quad\quad c\,(\\
\quad\quad\quad\quad\quad m2\,(Xyz(5, Xyz(10, Xyz(15, Zyx))), m1, m1)\,, \\
\quad\quad\quad\quad\quad m1 \\
\quad\quad\quad\quad)\, \\
\quad\quad\quad)\, \\
\quad\quad)\, \\
\quad)\,, \\
\quad n1 \\
) \rightarrow Xyz(2, Xyz(1, Xyz(3, Xyz(11, Xyz(8, Xyz(5, Xyz(10, Xyz(15, Zyx)))))))))
\end{array}
$$

The output code is as follows:

```
let rec flatten_list = function
  | Zyx -> Zyx
  | Xyz (h, t) -> Xyz (h, flatten_list t)

let rec flatten_tree = function
  | m1 | n1 -> Zyx
  | c (m2 (lst, left), right) | c (left, n2 (lst, right)) ->
      let lst' = flatten_list lst in
```

```
      let left' = flatten_tree left in
      let right' = flatten_tree right in
      append_list lst' (append_list left' right')

and append_list list1 list2 =
  match list1 with
  | Zyx -> list2
  | Xyz (h, t) -> Xyz (h, append_list t list2)
```

This way we access the LLM's, code synthesizing capability rather than its recall capability.

4 Results

We compare the results across the benchmarks used in traditional program synthesis works. Table-1 presents the benchmarks. The ✓ sign means that the benchmark was synthesized by the technique. NA(Not Applicable) represents that the technique was not tested against that benchmark. Lastly, ✗ means that the model was not able to synthesize the benchmark correctly.

The selected LLM model used for the results analysis is derived from Experiment-3.

Table 1: Program Synthesis Benchmarks

Benchmark	**Lambda Square**	**Synquid**	**Smyth**	**Burst**	**Leon**	**LLM**
list_add	✓	NA	NA	NA	NA	✓
list_append	✓	NA	✓	✓	NA	✓
list_concat	✓	✓	NA	NA	NA	✓
list_dedup	✓	✓	NA	NA	NA	✓
list_droplast	✓	NA	NA	NA	NA	✓
list_dropmax	✓	NA	NA	NA	NA	✓
list_dupli	✓	✓	✓	✓	NA	✓
list_evens	✓	NA	NA	NA	NA	✓
list_length	✓	✓	✓	✓	NA	✓
list_max	✓	NA	NA	NA	NA	✓
list_member	✓	✓	NA	NA	NA	✓
list_multfirst	✓	NA	NA	NA	NA	✓
list_multlast	✓	NA	NA	NA	NA	✓
list_reverse	✓	✓	NA	NA	NA	✓
list_shiftl	✓	NA	NA	NA	NA	✓
list_shiftr	✓	NA	NA	NA	NA	✓
list_sum	✓	✓	✓	✓	NA	✓
list_isempty	NA	✓	NA	NA	NA	✓
list_replicate	NA	✓	NA	NA	NA	✓
list_concatListOfLists	NA	✓	✓	✓	NA	✓

list_takeFirstN	NA	✓	✓	✓	NA	✗
list_dropFirstN	NA	✓	✓	✓	NA	✓
list_delete	NA	✓	NA	NA	✓	✓
list_map	NA	✓	✓	✓	NA	✗
list_zip	NA	✓	NA	NA	NA	✓
list_zipWithFunc	NA	✓	NA	NA	NA	✗
list_cartesianProduct	NA	✓	NA	NA	NA	✓
list_ithElement	NA	✓	NA	NA	NA	✓
list_indexEle	NA	✓	NA	NA	NA	✓
list_insertEnd	NA	✓	NA	NA	NA	✓
list_foldr	NA	✓	✓	✓	NA	✗
list_appendUsingFold	NA	✓	NA	NA	NA	✗
list_compress	NA	NA	✓	✓	NA	✓
list_evenParity	NA	NA	✓	✓	NA	✓
list_filter	NA	NA	✓	✓	NA	✓
list_head	NA	NA	✓	✓	NA	✓
list_last	NA	NA	✓	✓	NA	✓
list_tail	NA	NA	✓	✓	NA	✓
list_pairwiseSwap	NA	NA	✓	✓	NA	✗
list_snoc	NA	NA	✓	✓	NA	✗
list_revAppend	NA	NA	✓	✓	NA	✗
list_revFold	NA	NA	✓	✓	NA	✗
list_revSnoc	NA	NA	✓	✓	NA	✗
list_revTailcall	NA	NA	✓	✓	NA	✗
list_insert	NA	✓	✓	✓	✓	✓
list_union	NA	NA	NA	NA	✓	✓
list_diff	NA	NA	NA	NA	✓	✓
list_split	NA	NA	NA	NA	✓	✓
list_adjDuplicates	NA	✓	NA	NA	NA	✓
tree_leafCount	✓	NA	✓	✓	NA	✓
tree_countNodes	✓	✓	✓	✓	NA	✓
tree_flatten	✓	NA	NA	NA	NA	✓
tree_height	✓	NA	NA	NA	NA	✓
tree_incrt	✓	NA	NA	NA	NA	✓
tree_leaves	✓	NA	✓	✓	NA	✓
tree_maxt	✓	NA	NA	NA	NA	✓
tree_member	✓	✓	NA	NA	NA	✓
tree_sum	✓	NA	NA	NA	NA	✓
tree_preorder	NA	✓	✓	✓	NA	✓
tree_inorder	NA	✓	✓	✓	NA	✓
tree_postorder	NA	✓	✓	✓	NA	✓
tree_nodesAtLevel	NA	NA	✓	✓	NA	✓

bst_isMember	NA	✓	NA	NA	NA	✓
bst_insert	NA	✓	✓	✓	NA	✓
bst_delete	NA	✓	✓	✓	NA	✓
bst_sort	NA	✓	NA	NA	NA	✓
heap_isMember	NA	✓	NA	NA	NA	✓
heap_insert	NA	✓	NA	NA	NA	✗
avl_rotateLeft	NA	✓	NA	NA	NA	✓
avl_rotateRight	NA	✓	NA	NA	NA	✓
avl_insert	NA	✓	NA	NA	NA	✗
avl_delete	NA	✓	NA	NA	NA	✗
avl_extractMin	NA	✓	NA	NA	NA	✗
avl_balance	NA	✓	NA	NA	NA	✗
rbt_balanceLeft	NA	✓	NA	NA	NA	✗
rbt_balanceRight	NA	✓	NA	NA	NA	✗
rbt_insert	NA	✓	NA	NA	NA	✗
nestedStruct_flatten	NA	✓	NA	NA	NA	✓
nestedStruct_append	NA	✓	NA	NA	NA	✗
nestedStruct_incrs	NA	✓	NA	NA	NA	✗
nestedStruct_join	NA	✓	NA	NA	NA	✗
nestedStruct_prepend	NA	✓	NA	NA	NA	✗
nestedStruct_replace	NA	✓	NA	NA	NA	✗
nestedStruct_sumNodes	NA	✓	NA	NA	NA	✓
nestedStruct_sums	NA	✓	NA	NA	NA	✓
nestedStruct_sumTrees	NA	✓	NA	NA	NA	✓
bool_band	NA	NA	✓	✓	NA	✓
bool_bor	NA	NA	✓	✓	NA	✓
bool_impl	NA	NA	✓	✓	NA	✓
bool_neg	NA	NA	✓	✓	NA	✓
bool_xor	NA	NA	✓	✓	NA	✓

The benchmarks which were not synthesized correctly by the LLM model fall into two categories. **Category-1** → The LLM was not able to infer the method/function correctly from the input-output examples. **Category-2** → The LLM was able to infer the method/function correctly but the code produced by it did the specify the logical specification depicted by input-output examples.

Let us categorize the failed benchmarks into the above two categories in the below Table-2. The ✓sign indicates that the failed benchmark falls in that particular cateogory while ✗ indicates that the benchmark does not fall in that category.

Table 2: Program Synthesis Benchmarks

Failed Benchmarks	**Category-1**	**Category-2**
list_takeFirstN	✓	✗

list_map	✓	✗
list_zipWithFunc	✓	✗
list_foldr	✓	✗
list_appendUsingFold	✓	✗
list_pairwiseSwap	✓	✗
list_snoc	✓	✗
list_revAppend	✓	✗
list_revFold	✓	✗
list_revSnoc	✓	✗
list_revTailcall	✓	✗
heap_insert	✗	✓
avl_insert	✗	✓
avl_delete	✗	✓
avl_extractMin	✗	✓
rbt_balanceLeft	✗	✓
rbt_balanceRight	✗	✓
rbt_insert	✗	✓
nestedStruct_append	✓	✗
nestedStruct_incrs	✓	✗
nestedStruct_join	✓	✗
nestedStruct_prepend	✓	✗
nestedStruct_replace	✓	✗

5 Discussion on Results

This section provides an in-depth analysis and interpretation of the obtained results, aiming to gain deeper insights into the implications and significance of the findings. In our evaluation, the LLM model was subjected to rigorous testing against a comprehensive set of 91 benchmarks. The results revealed that the LLM successfully synthesized the correct program for the majority of the benchmarks. However, it encountered challenges with 23 benchmarks, where it was unable to generate the desired program. Among these 23 failed benchmarks, it struggled to infer the correct method or function for 14 of them. Additionally, for the remaining 11 benchmarks, it inferred the correct method or function but generated incorrect programs during the synthesis process.

Let's analyze the results on a case-by-case basis.

Lists: There are several methods which were not inferred correctly by input-output examples as shown in Table-2. The reason might be that the task of inferring a function's implementation solely from input-output examples typically requires a sufficient number of diverse examples to correctly generalize the function's behavior. It is possible that the examples provided were not diverse enough, leading to an inability to infer the correct behavior. Also, methods like list_map, list_foldr, list_appendUsingFold, and list_revFold are higher-order functions, which means they take

other functions as arguments. Inferring these from examples is especially difficult because it involves not only understanding the list manipulations but also figuring out the behavior of the function passed as argument. This complexity can make it challenging for the LLM to infer the methods correctly.

Heap: The heap_insert benchmark which was incorrectly generated by LLLM, is an operation that is a non-trivial task that involves correctly maintaining the heap's properties after insertion. So, one of the reasons which lead to incorrect code generation for this benchmark might be that the training data for the LLM lacked enough examples of OCaml programming with heap.

AVL Trees: AVL trees are self-balancing binary search trees, and operations like insert, delete, and extracting the minimum element are implemented using recursion. So, the algorithms to correctly perform these operations and maintain the AVL tree properties are quite complex. Therefore, the complex nature of these AVL operation algorithms might have been a barrier in synthesizing the correct code.

Red Black Trees: The operations on red-black trees typically involve recursion and multiple conditions that need to be checked and handled (like the color of nodes and balance of the tree). This combination of recursion and condition handling could have be challenging for the LLM.

Nested Data Structures: For nested data structures, methods shown in Table-2 were not inferred correctly. Inferring methods like nestedStruct_append, nestedStruct_incrs, nestedStruct_prepend, and nestedStruct_replace from input-output examples would require the model to understand how these operations are applied at each node in the tree structure and is a non-trivial task. Adding to this complexity is the custom definition of both the list and tree structures. Moreover, the binary tree of lists was defined using recursive data structures with multiple constructors ($m1$, $m2$, $n1$, $n2$). This is a significantly more complex representation than a simple binary tree. Given these custom types, the LLM would need to infer not just the operations but also the pattern matching necessary to traverse and manipulate the data structures correctly. This might be a reason for incorrect inferring of these methods.

6 Conclusion

This research has explored the use of large language models (LLMs), particularly chatGPT, for synthesizing functional programs from high-level specifications and input-output examples. Evaluations were conducted against a comprehensive set of benchmarks, shedding light on the capabilities and limitations of using LLMs in comparison to traditional program synthesis techniques. The results of this study reveal that while LLMs can successfully synthesize the correct program for most benchmarks, challenges arise in more complex scenarios, particularly when dealing with higher-order functions, intricate data structures such as trees and nested structures, and customly defined data types. Furthermore, issues emerged when inferring methods based solely on input-output examples, emphasizing the need for diverse and ample examples to effectively generalize the intended behavior. This research work of diving into leveraging of LLMs for program synthesis task paves the way for future research aimed at overcoming these challenges.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. *International Conference on Computer Aided Verification*, 2013.

[2] Etienne Kneuss, Viktor Kuncak, Ivan Kuraj, and Philippe Suter. Synthesis modulo recursive functions. *ACM SIGPLAN Notices*, 48(10):407–426, 2013.

[3] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *PLDI*, 2015.

[4] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *PLDI*, 2016.

[5] JUSTIN LUBIN, NICK COLLINS, CYRUS OMAR, and RAVI CHUGH. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages*, 4:1–29, 2020.

[6] ANDERS MILTNER, ADRIAN TREJO NUÑEZ, ANA BRENDEL, SWARAT CHAUDHURI, and ISIL DILLIG. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages*, 6(21):1–29, 2022.