RedString: A Dynamic Pseudocode Interface Backed by Generative Language Models

Arushi Somani

Contents

1 Introduction

The recent exponential development and adoption of programming assistants like Codex and Github Copilot raises the question of how these generative assistants best interact with the human developers. The fundamental goal of these assistants is to abstract away the "how", and allow the programmer to focus on the "what". Through this lens, we analyze two different state of the art interfaces for synthesis: code suggestion using auto-completion, and code generation using Programming by Example/Demonstration.

For auto-complete suggestion generators like Copilot, they are excellent tools for writing boilerplate code. However, they bear two critical flaws:

1. **Lack of Abstraction**: There is no abstraction created between the programmer and the programming assistant. The power of these suggestion tools is to suggest code the user was thinking about writing and not necessarily code that the user does not need to worry about writing.

2. **Lack of Guarantees of Correctness**: There are no guarantees about the correctness of the generated code. Once an autocomplete suggestion is accepted, the programmer either needs to verify it using their own reasoning and keep coding, or the programmer needs to halt their code-writing, develop a small testing framework, test the generated code, and then continue.

Programming by example and programming by demonstration tools do create an abstraction between the programmer and the AI, and they have correctness baked within the design. However, they bear other flaws that we elucidate here:

- **Synthesis Duration**: These assistants often take a long time to synthesize programs. This duration increases as the complexity of the I/O or demonstration increases. It also abstracts away the entirety of the synthesis process, such that users develop frustration around re-specifying the problem repeatedly because of ambiguous specifications.
- **Difficulty of Generating Examples**: For problems like PBE, it is sometimes notoriously difficult to generate examples — this is because a lot of the program might have complex inputs and outputs like entire files or multiple levels of user input. Such a technique only works in specific domains.

In this project, we suggest a new interface that allows programmers to focus solely on the logic underlying their task. We propose the following concepts, answering the following questions:

1. **Enforcing Abstraction**
    - How could we encourage the human developers to best focus on the "what", and abstract away the "how" of programming?
    - When synthesis fails, how can we allow the programmer to increase the granularity of the specification to better assist the synthesis engine?
2. **Feedback and Conversation with the User**
    - How could we develop trust between the user and the synthesized code, such that the user could rely on the generated code blocks without needing to manually read through them or write a test bench for them?
    - How could we allow the user to test the code without needing to write detailed and exhaustive I/O examples?
    - How can we integrate feedback about corrections to the program received from the oracle into our generation step?

We present RedString, an interface for programming, built on Python and backed by Transformer based generative models, that allows users to synthesize functions, test them, and compose them together to form larger programs.

2 Related Work

There has been previous work that performs usability testing studies on using Large Language Models in programming [1] [2]. Our project is a derivative of these, and will build up upon these projects. These has also been some work in developing new interfaces for LLMs [3]. However, the core of these works is the underlying models instead of the interface that exposes them. Some work [4] has attempted to expose a novel interface to these models. However, this work limits itself to visual domains like the web and HTML domain. Our project develops a general interface for all functional programming problems.
With how recent these LLM tools are, there has not been much investigation into the best way of using them, despite long-standing trends.

3 Planned Approach

We suggest two core techniques to solve the questions and concepts suggested above.

3.1 Enforcing Abstraction through Dynamic Abstraction Barriers

Programming is inherently recursive — functions call functions which call functions. We use this inherent design of programs to create *dynamic abstraction barriers*. The user defines functions at a high-level in a pseudo-code like fashion, and then relies on the LLM to define the function that is used.
We elaborate with an example.
Let's pretend that the user were to write something like this:

```
def tic_tac_toe():
    board = init_board()
    while True:
        board = player_x_takes_action(board)
        winner = get_winner(board)
        if winner is not None:
            announce_winner(winner)
            return
        board = player_o_takes_action(board)
        winner = get_winner(board)
        if winner is not None:
            announce_winner(winner)
            return
```

After this, the user would rely on the LLM to define `init_board` and `player_x_takes_action` and so on and so forth, for every function that is used within `tic_tac_toe`.

What if the synthesis engine is not able to come up with a correct function (say, `player_x_takes_action`) The user would then increase the granularity of their specification. They would additionally specify:

```
def player_x_takes_action(board):
    while True:
        action = get_action_from_user()
        if can_make_action(action, board):
            return edit_board_on_action(board, action)
        else:
            print("Incorrect input, try again")
```

This increases the work that the user has to do, but *decreases* the load upon the synthesis engine to make crucial decisions — like having to take input from the user, for example. This method takes code-writing from suggestion-design to instead a conversation between the user and the synthesis tool, where the errors of the synthesis tool request the user to provide more details.

3.2 Receiving Feedback Through Oracle-Guidance

It is commonly known that large language models work best at high values for k when used under $Pass@K$ models [3]. We use three insights to develop a feedback mechanism for our synthesis tool:

1. Since the function written by the synthesis engine is a complete module, we can evaluate it to make sure that it does not error on simple computation.

2. Since we have the call to that function, if we were to get an example of input/output values from the user, we could trivially execute the function to make sure there are no run-time errors.

3. Once we have a syntactically correct function, if we were to characterize the types of inputs, we could "fuzz" the function, and represent every branch taken by one input/output example.

Thus, in our model, we initially request a single characteristic input example from the user (if applicable). If the user is synthesizing many functions, we infer the input example from the pseudo-code structure. In the `tic_tac_toe` example, we can infer the input to `player_x_takes_action` ("board") as the output from `init_board` — which we would already have synthesized.

The core idea here is that, if we could get the output of a function checked off, we could use it as a reliable input to the next synthesized function. Through this structure, the user would be able to "sanity test" the functions generated manner for their correctness.

However, just a single sanity test might not be enough to win over the trust of the user. We want to show the user one iteration of every possible output type. To do this, we employ modern fuzzing techniques. We could use concolic execution techniques to come up with random examples that give us complete branch coverage. Since the generated module is necessarily small, this would not be inefficient. We also would be happy with any set of examples that span the branches of the program, which loosens the constraints of generating this set.

The only condition here would be that we wish to generate **readable** examples. Here, we posit an assumption that, in functional programming, the readability of the input is directly related to its size. Thus, we will bias our fuzzing model towards generating smaller inputs, which we posit would be a decent heuristic for interpretable I/O examples generated.

4 Success Criteria

Lastly, we talk about what our evaluation phase would be and what "success" would look like for this project. Our evaluation phase would include presenting a prototype of our synthesis interface to users, presenting them with code problems to solve, and judging by the following metrics:

1. Whether they were able to solve a problem they can solve without any code assistant tools

2. Whether they are able to solve a problem they cannot solve on their own

3. Whether the time taken to solve the problem decreases when using our prototype versus when programming on their own

4. Whether the time taken to solve the problem decreases when using our prototype versus when programming with a SOTA assistant like Github Copilot

Our hypothesis states that this interface would allow programmers to solve more problems that they would be able to completely on their own, in lesser time.

References

[1] Barke, Shraddha, et al. "Grounded Copilot: How Programmers Interact with Code-Generating Models." ArXiv:2206.15000 [Cs], 2 Aug. 2022, arxiv.org/abs/2206.15000.

[2] Jiang, Ellen, et al. "Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models." CHI Conference on Human Factors in Computing Systems, 29 Apr. 2022, 10.1145/3491102.3501870. Accessed 27 Sept. 2022.

[3] Chen, Mark, et al. "Evaluating Large Language Models Trained on Code." ArXiv:2107.03374 [Cs], 14 July 2021, arxiv.org/abs/2107.03374.

[4] "GenLine and GenForm: Two Tools for Interacting with Generative Language Models in a Code Editor." The Adjunct Publication of the 34th Annual ACM Symposium on User Interface Software and Technology, 10 Oct. 2021, 10.1145/3474349.3480209. Accessed 27 Sept. 2022.