Title: Neurosymbolic API Synthesis: Improving Code Generation through Hybrid Prompting

1. Problem Statement: Generating code that correctly uses complex APIs or libraries remains challenging for language models, especially when dealing with large, poorly documented, or rapidly evolving APIs. This problem is particularly acute in real-world software development scenarios where developers need to interact with diverse and complex APIs.

2. Motivation: Current approaches often rely on fine-tuning on API-specific datasets or using retrieval-augmented generation, which can be data-intensive and may not generalize well to unseen APIs. By combining neural generation with symbolic reasoning about API structures and constraints, we can potentially create a more robust and generalizable approach to API-aware code generation. This hybrid approach leverages the strengths of both neural and symbolic methods, potentially leading to more accurate and reliable code generation across a wide range of APIs.

3. Proposed Method: We introduce Neurosymbolic API Synthesis, a hybrid prompting technique that integrates neural generation with symbolic API reasoning. The method consists of the following steps:

(1) API Structure Extraction: Prompt the model to extract a symbolic representation of the API's structure, including types, functions, and their relationships.

(2) Neurosymbolic Generation:

a. Neural suggestion of API usage patterns

b. Symbolic type checking and constraint propagation

c. Neural refinement based on symbolic feedback

(3) Iterative Refinement: Repeat step 2 until a valid and efficient API usage pattern is synthesized.

(4) Final Code Generation: Prompt the model to generate complete code that adheres to the synthesized API usage pattern while solving the original problem.

4. Step-by-Step Experiment Plan:

Step 1: Dataset Preparation: Collect a diverse set of coding tasks involving complex APIs from popular libraries in multiple programming languages. Focus on APIs from libraries such as TensorFlow, PyTorch, Pandas, and Scikit-learn for Python, and Spring Framework, Apache Hadoop, and Java Collections for Java. Ensure the dataset covers a range of task complexities and API usage patterns.

Step 2: Baseline Implementation: Implement and evaluate baseline methods:

a. Direct prompting: Simply ask the model to generate code for the given task.

b. Few-shot prompting: Provide a few examples of correct API usage before asking the model to generate code.

c. Chain-of-thought prompting: Ask the model to explain its reasoning step-by-step while generating code.

Step 3: Neurosymbolic API Synthesis Implementation: Implement the proposed method:

a. API Structure Extraction: Prompt the model with: "Given the following API documentation, extract a structured representation of the API, including types, functions, and their relationships: [API documentation] Provide the structured representation in JSON format."

b. Neurosymbolic Generation:

- Neural suggestion: "Suggest an API usage pattern for the following task: [Task description] Based on the API structure: [Extracted API structure]"

- Symbolic checking: Implement a rule-based system to check type consistency and API constraints.

- Neural refinement: "Refine the following API usage pattern based on these constraint violations: [API usage pattern] [Constraint violations]"

c. Iterative Refinement: Repeat the neurosymbolic generation step until no constraint violations are found or a maximum number of iterations is reached.

d. Final Code Generation: "Generate complete code that solves the following task using the synthesized API usage pattern: [Task description] [Synthesized API usage pattern]"

Step 4: Model Selection: Use GPT-4 and GPT-3.5-turbo from OpenAI's API for all experiments. Additionally, include Claude-3.5 from Anthropic and Gemini from Google as alternative models for comparison.

Step 5: Evaluation: Evaluate the generated code on the following metrics:

a. Compilation success rate: Percentage of generated code that compiles without errors.

b. Runtime correctness: Percentage of generated code that produces correct output for given test cases.

c. API usage correctness: Manual evaluation of whether the generated code uses the API correctly and efficiently.

d. Code quality: Use automated tools like Pylint for Python and PMD for Java to assess code quality.

e. Generalization: Test the method on APIs not seen during the initial evaluation to assess generalization capability.

Step 6: Comparative Analysis: Compare the performance of the Neurosymbolic API Synthesis method against the baselines across all metrics. Conduct statistical significance tests to validate the improvements.

Step 7: Ablation Studies: Perform ablation studies to understand the contribution of each component:

a. Remove the API structure extraction step

b. Remove the symbolic checking step

c. Vary the number of iterations in the refinement process

Step 8: Error Analysis: Analyze cases where the proposed method fails or performs worse than baselines. Categorize error types and identify potential areas for improvement.

5. Test Case Examples:

Test Case 1:

Baseline Prompt Input (Direct Prompting): Write a Python function that uses the TensorFlow library to create a simple neural network for binary classification with one hidden layer. The function should take the number of input features, hidden units, and output units as parameters.

Baseline Prompt Expected Output (Direct Prompting):

import tensorflow as tf

def create\_neural\_network(input\_features, hidden\_units, output\_units):

model = tf.keras.Sequential([

tf.keras.layers.Dense(hidden\_units, activation='relu', input\_shape=(input\_features,)),

tf.keras.layers.Dense(output\_units, activation='sigmoid')

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

return model

Proposed Prompt Input (Neurosymbolic API Synthesis; Step 1: API Structure Extraction): Given the following TensorFlow API documentation, extract a structured representation of the API, including types, functions, and their relationships for creating a simple neural network:

tf.keras.Sequential: Creates a sequential model.

tf.keras.layers.Dense: Adds a dense (fully connected) layer to the model.

tf.keras.Sequential.compile: Configures the model for training.

Provide the structured representation in JSON format.

Proposed Prompt Expected Output (Neurosymbolic API Synthesis; Step 1: API Structure Extraction):

{

"tf.keras.Sequential": {

"type": "class",

"methods": ["compile"],

"parameters": []

},

"tf.keras.layers.Dense": {

"type": "class",

"parameters": ["units", "activation", "input\_shape"]

},

"tf.keras.Sequential.compile": {

"type": "method",

"parameters": ["optimizer", "loss", "metrics"]

}

}

Proposed Prompt Input (Neurosymbolic API Synthesis; Step 2: Neural Suggestion): Suggest an API usage pattern for creating a simple neural network for binary classification with one hidden layer using TensorFlow. The function should take the number of input features, hidden units, and output units as parameters. Base your suggestion on the following API structure:

[Include the API structure from Step 1]

Proposed Prompt Expected Output (Neurosymbolic API Synthesis; Step 2: Neural Suggestion):

1. Create a tf.keras.Sequential model

2. Add a tf.keras.layers.Dense layer with hidden\_units and 'relu' activation, specifying input\_shape

3. Add another tf.keras.layers.Dense layer with output\_units and 'sigmoid' activation

4. Compile the model using the compile method with appropriate optimizer, loss, and metrics

Proposed Prompt Input (Neurosymbolic API Synthesis; Step 3: Symbolic Checking): The symbolic checking step would be implemented as a rule-based system in the code, not as a prompt. It would verify type consistency and API constraints based on the extracted API structure and the suggested usage pattern.

Proposed Prompt Input (Neurosymbolic API Synthesis; Step 4: Neural Refinement): Refine the following API usage pattern for creating a simple neural network using TensorFlow, addressing these constraint violations:

[Include the API usage pattern from Step 2]

Constraint violations:

- input\_shape parameter in the first Dense layer should be a tuple

- compile method requires specific types for optimizer, loss, and metrics parameters

Proposed Prompt Expected Output (Neurosymbolic API Synthesis; Step 4: Neural Refinement):

1. Create a tf.keras.Sequential model

2. Add a tf.keras.layers.Dense layer with hidden\_units and 'relu' activation, specifying input\_shape as a tuple (input\_features,)

3. Add another tf.keras.layers.Dense layer with output\_units and 'sigmoid' activation

4. Compile the model using the compile method with optimizer='adam', loss='binary\_crossentropy', and metrics=['accuracy']

Proposed Prompt Input (Neurosymbolic API Synthesis; Step 5: Final Code Generation): Generate complete Python code that creates a simple neural network for binary classification with one hidden layer using TensorFlow, based on the following synthesized API usage pattern:

[Include the refined API usage pattern from Step 4]

The function should take the number of input features, hidden units, and output units as parameters.

Proposed Prompt Expected Output (Neurosymbolic API Synthesis; Step 5: Final Code Generation):

import tensorflow as tf

def create\_neural\_network(input\_features, hidden\_units, output\_units):

model = tf.keras.Sequential([

tf.keras.layers.Dense(hidden\_units, activation='relu', input\_shape=(input\_features,)),

tf.keras.layers.Dense(output\_units, activation='sigmoid')

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

return model

Explanation: The Neurosymbolic API Synthesis method produces a more robust and correct implementation by leveraging API structure knowledge and iterative refinement. It correctly specifies the input\_shape as a tuple and provides appropriate parameters for the compile method, which might be missed in a direct prompting approach.

6. Fallback Plan: If the proposed Neurosymbolic API Synthesis method does not significantly outperform baselines, we can pivot the project in several ways. First, we could conduct an in-depth analysis of where and why the method fails, potentially uncovering interesting insights about the limitations of current language models in understanding and using complex APIs. This could involve categorizing error types, analyzing the quality of extracted API structures, and examining the effectiveness of the symbolic checking step. Second, we could explore variations of the method, such as incorporating retrieval-augmented generation to supplement the API structure extraction step, or experimenting with different prompting strategies for each step of the process. Finally, we could shift focus to developing a benchmark dataset for evaluating API-aware code generation, which would be valuable for the broader research community regardless of our method's performance.