Multi-modularizing CodeChain による コード生成タスクの細分化が精度に与える影響

井上貴之 [1]　鶴岡慶雅 [2]
[1] 東京大学工学部電子情報工学科　[2] 東京大学大学院情報理工学系研究科電子情報学専攻
{inoue, tsuruoka}@logos.t.u-tokyo.ac.jp

概要

大規模言語モデルによる競技プログラミングのような入力に対し出力が一意に定まるコード生成においては、タスクを小タスクの連続に分割し、それぞれを解決するアプローチが取られている。本研究ではコード内定義関数を利用しタスクの細分化を行う CodeChain と呼ばれる機構について、高難易度の問題における精度向上に向け、関数内関数を利用しさらなる細分化を行うようプロンプトの改修および関数内関数の抽出、評価を追加した。結果、一部のデータ、モデルでは精度の向上が確認できた。

1　はじめに

大規模言語モデル (Large Language Model, LLM) は大量のテキストデータで構成されたコーパスを入力データとして学習した機械学習モデルである。2017 年における Attention 機構を導入した Transformer アーキテクチャの提唱 [1]、その翌年の Transformer を導入した学習モデルである GPT [2]、BERT [3] の開発を端緒として LLM の開発が進み、よりパラメータの大きく、高精度なモデルが登場した [4, 5, 6]。LLM は汎用性が高く、文章の補完や生成にとどまらずコードの生成や論理的推論、知識問題への応答が可能となっている。また、転移学習やファインチューニングの利用により StarCoder [7] や WizardCoder [8] などコード生成に特化した LLM モデルも出現した。

しかし、LLM によるコード生成には依然として課題が多く、HumanEval [9] や MBPP [10] ベンチマークなどといった評価指標の問題群は初歩的な内容に留まっている。より複雑なタスクをクリアできるプログラムの生成においては、タスク達成までをより小さなタスクの連続へ分割し、その達成方法をそれぞれに与える計画の立案力や出力を修正する能力が必要になる。

上記の課題に対し、CodeChain [11] は、関数定義を用いてタスクの細分化を行い、出力から選別された関数を利用して再生成することで出力を修正している。しかし、その精度は APPS ベンチマーク [12] において最高でも GPT-3.5 [13] と併用した場合の 30.24%である。特に最高難易度の “competition“の問題群においては精度は 12.38 %にとどまる。そこで、本研究では細分化されたタスクをそれぞれ細分化する手法 [14, 15] を用いてタスク一つ一つの粒度を高め、精度の向上を目指す。

2　CodeChain

本節では本研究の土台となる CodeChain について説明する。

本研究で扱う問題群は競技プログラミングを想定した、入力に対し出力が一意に定まる問題である。問題は説明文、入力変数の型および制約、出力変数の型および制約、入出力の一例が与えられる。競技プログラミングには、コード提出前に参照できる入出力対が存在する。コード生成タスクではこれに準え、コード生成の過程上で参照できる入出力対の Public test データと最終出力を評価する入出力対である Private test データが与えられている。

続いて全体の概要を図 1 に示す。はじめに、以下の 4 つの手順を 1 サイクルとして定義する: (1) Chain of Thought [16] を用いたプロンプトによる関数を用いたタスクの細分化 (2) Public Test データによる出力コードの評価 (3) テストケースを通過した出力の関数部分の抽出 (4) K-means クラスタリングによる代表関数の抽出および (1) のプロンプトへの代表関数の追記 この一連の手順を 1 サイクルとして所定の回数行ったのち、再度プロンプトに代表関数を加え生成を行ったものを最終出力として扱う。以下、(1) から詳細を説明する。

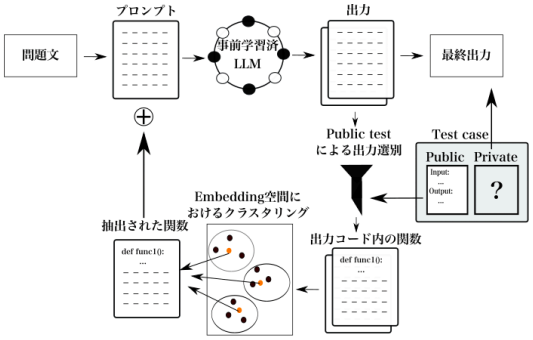

図 1: CodeChain

CodeChain は課題となる問題および出力として与えられるコードの対を One-shot [17] で与える。出力コードはコード 1、2 のように二段階に分かれており、STEP 1 では出力コードに使用する関数について関数名、機能、入出力の型のみ定義する。STEP 2 では STEP 1 で記載した関数の詳細を実装し、これを用いて問題を解くコードを構成する。これにより解決すべきタスクが関数の個々の機能実装と統合に分割される。CodeChain においては Public test を用いて各サイクルにおける出力を評価し、すべてのテストケースを通過した出力のみを選別する。

ソースコード 1: STEP 1 の例

```
STEP 1: GENERATE SUB-MODULES:
```module
def count_start_end_chars(words):
    """
    Description: This function counts the number of
        words that start and end with each character.
    Input:
    words (list): A list of binary words.
    Output:
    start_count (defaultdict): A dictionary containing
        the count of words that start with each
        character.
    end_count (defaultdict): A dictionary containing
        the count of words that end with each
        character.
    """
```
...
```

ソースコード 2: STEP 2 の例

```
STEP 2: GENERATE PYTHON CODE
```python
import collections
from utils import *

def count_start_end_chars(words):
    start_count = collections.defaultdict(int)
        end_count = collections.defaultdict(int)
        for word in words:
            start_count[word[0]] += 1
            end_count[word[-1]] += 1
        return start_count, end_count

...

t = int(input())
for _ in range(t):
    n = int(input())
    words = []
    for _ in range(n):
        words.append(input())
    total_reversed, reversed_words = solve_task(words)
    print(total_reversed)
    if total_reversed != 0:
        print(*reversed_words)
```
```

上記のテストケースを通過したコードから関数実装部分のみを切り出す。取り出された関数はエンコーダを有する事前学習済み LLM を用いて実数値ベクトルへ変換を行う。エンコーダとして用いられたモデルは CodeBERT [18]、CodeT5+ [19]、StarCoder [7] の 3 種類から選ばれる。本実験では StarCoder に統一する。

ベクトル表現へ変換された関数群は K-means 法によりクラスタリングを実行し、類似している関数のうち代表となるもののみを抽出する。クラスタ k における代表値 C_k は、クラスタ内 i 番目の要素を S_i^k、クラスタ内要素の平均を μ_k として以下の式 1 の通り決定される。

抽出された関数はプロンプトに追記される。これを用いて再度生成、評価、関数抽出を繰り返す。

$$C_k = \underset{S_k}{\operatorname{argmin}} ||S_i^k - \mu_k|| \tag{1}$$

3 提案手法

3.1 関数の多重分解

続いて本研究で構成する Multi-modularizing CodeChain の主軸である関数の多重分解の手法について説明する。CodeChain では、コード全体で達成すべきタスクを特定の機能を果たす関数を複数定義することで個々の関数の実装にタスクを分割する。この関数実装をさらに複数の関数内関数によって細分化することで複雑なタスクを簡単なタスクの集合として捉えることを目指す。

本研究では、関数内関数の利用により一度の生成でタスクの多段階分解を行う。コード 3 に示すようにある関数の定義記述内部で再度関数を定義する文構造をプロンプトに導入する。これにより生成時に必要に応じて関数内関数の定義においても出力例を利用して関数内関数が生成されるため、必要に応じ関数の果たすべきタスクを細分化できる。

ソースコード 3: 関数内関数を含む定義の例

```
def solve_task(words):

    def is_reversed(s):
        return s == s[::-1]

    def count_reversed_words(words):
        count = 0
...
```

3.2 Multi-modularizing CodeChain

図 2 に全体図を示す。コード 4 に変更したプロンプトを示す。赤字部分が CodeChain からの変更点である。新たに関数内関数についても機能の記述を行う STEP を追加し、全 3 S TEP の構成となっている。出力例には関数内関数を含む関数と含まない関数の両方を与え、関数内関数を必ずしも含まないようにする。入力例は CodeChain と共通で APPS train データの問題番号 0 の問題を用いる。プロンプトに与える出力例はコード 5 に記載する。そして前節で説明した通り、テストケースを通過したコードから関数、関数内関数を個別に抽出、クラスタリングを行う。クラスタ数は各サイクルにおいてそれぞれ 5、4、3、2、1 とする。

ソースコード 4: 入力プロンプトの雛形

```
    Below is a competitive programming question. Read
        the question carefully.

*Instruction*
Develop a well-structured Python solution for the
    provided problem that obeys the constraints and
    passes the example test cases. Ensure modularity
    and considering potential edge cases and failures.
     Start by outlining the required code modules,
    including function headers and signatures.
    Subsequently, proceed to implement each module to
    create the final code.

In simpler terms, create a clean and organized Python
    solution for the given problem. Break it down into
     smaller parts (modules) with clear function names
     and input/output specifications. Once the
    structure is ready, write the actual code for each
     module to complete the solution.

The output code needs to read from and write to
    standard IO. Please wrap your code answer using
    ```.

Example 1
TASK:
<<Example_problem>>
RELEVANT FUNCTIONS:
<<modules>>
RELEVANT SUB-FUNCTIONS:
<<sub-modules>>
RESPONSE:
STEP 1: GENERATE MODULES:
<<example generated modules>>
STEP 2: GENERATE SUB-MODULES:
<<example generated sub-modules>>
STEP 3: GENERATE PYTHON CODE
<<example generated code>>

Example 2
TASK:
<<new problem>>
RELEVANT FUNCTIONS:
<<modules>>
RELEVANT SUB-FUNCTIONS:
<<sub-modules>>
RESPONSE:
```

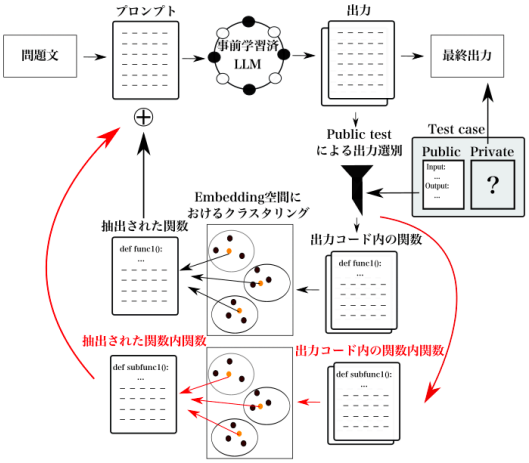


図 2: Multi-modularizing CodeChain

## 4 実験

### 4.1 実験設定

前節で説明した Multi-modularizing CodeChain を用いて出力の精度検証を行う。使用するモデルはクローズドモデルから GPT-3.5-turbo-16k [13]、オープンソースモデルから WizardCoder-15B-V1.0 を用いる。WizardCoder は VLLM フレームワーク [20] を使用してパラメータを利用する。Temperature (どの程度ランダムに応答するかを定める 0〜1 のパラメータ) は先行研究 [11] に倣い 0.6 とする。最大出力長は 8192 とする。出力コードについては 5 応答/入力 × 4 入力の形式で収集し、Pass@$k$ を算出して比較する。 Pass@$k$ は一般に以下の式 2 で算出される。

$$\text{Pass@}k = \mathbb{E}\left[1 - \frac{{}_{n-c}C_k}{{}_nC_k}\right] \tag{2}$$

($n$ : 出力コード数, $c$ : 正解した出力コード数 )

使用する問題データは APPS および CodeContests [21] の test データセットを用い、APPS については難易度ごとに精度を、各モデルに対して 3 回の出力の平均をとって比較する。APPS の test データは 5000 問あり、出力生成に長時間を要した。したがって 1 回分の取得データに対し、同データ内からより小さいサイズの問題群を取り出し、その誤差を推定する。絶対誤差は取り出した問題に応じて変動するため、同一モデルのにおける相対誤差について同一データ内であればどの部分を取り出しても一定であると仮定し、test データから難易度ごとに 50 問ずつランダムに選び、その出力結果の相対誤差を算出する操作を 3 回行い、そのうち最大値を出力の相対誤
```

差として計算した。

4.2 実験結果

実験結果を表 1、2 に示す。CodeChain および各モデルの機構を介さない生成の結果は先行研究 [11] から引用した。APPS については全ての難易度で有意な精度低下が確認され、CodeContests では、WizardCoder を用いた場合は Validation data における pass@1 を除き有意な精度の向上が確認され、GPT-3.5-turbo-16k においては有意な精度低下が確認された。

表 1: APPS test データにおける精度比較

Model	Introductory	Interview	Competition	All
WizardCoder-15B	26.04	4.21	0.81	7.90
+ CodeChain	26.29	7.49	3.75	10.50
+ Multi-modularizing CodeChain	16.29 ± 1.47	4.81 ± 0.43	1.4 ± 0.35	6.44 ± 0.78

表 2: CodeContests における精度比較

Model	Valid		Test	
	Pass@1	Pass@5	Pass@1	Pass@5
WizardCoder-15B	1.11	3.18	1.98	3.27
+ CodeChain	2.35	3.29	2.48	3.30
+ Multi-modularizing CodeChain	2.26 ± 0.30	**3.90 ± 0.33**	**3.23 ± 0.34**	**5.67 ± 0.34**
GPT-3.5-turbo-16k	6.81	16.23	5.82	11.16
+ CodeChain	12.86	16.91	10.27	14.11
+ Multi-modularizing CodeChain	8.51 ± 0.15	13.90 ± 0.33	6.81 ± 0.65	11.55 ± 0.65

4.3 考察

APPS、CodeContests の両方で CodeChain よりも精度が低下した要因については、Multi-modulariing CodeChain において、途中までは精度が向上したが最終サイクルで精度が下降したことが挙げられる。 図 3 は CodeContests における Multi-modularizing CodeChain (折れ線) と CodeChain の最終出力値 (直線) の比較である。実際の出力コードにおいて STEP 1、STEP 2 において宣言された関数、関数内関数と STEP 3 で実装された関数が一致していないものが見られた。STEP 1、STEP 2 の構造が類似しているため両者の混在が生じており、これがサイクルを通して精度が安定化しない理由であると考える。

図 4 は APPS における最終出力において、Private test を通過したコードにおける関数内関数の数を集計結果である。その平均は 1.35 個であり、関数内関数を有していたコードの割合は 28.59%であった。Multi-Modularizing CodeChain はコード全体の生成を 1 度に実行させるため関数内関数の生成可否を明示的に指示していないため、一部コードでは多段階なタスク分解が成功しているもののその利用率は低く、確認された精度向上も僅かであったと考える。

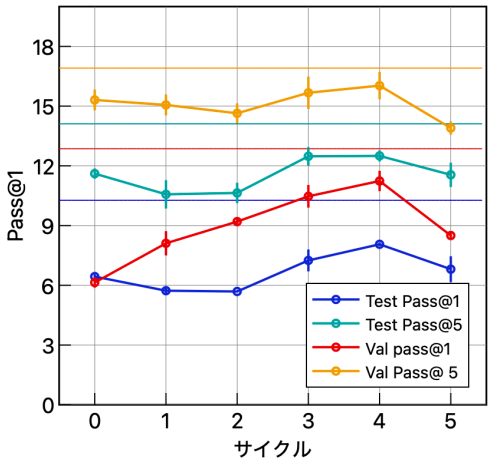

図 3: 精度推移比較 (モデル: GPT-3.5)

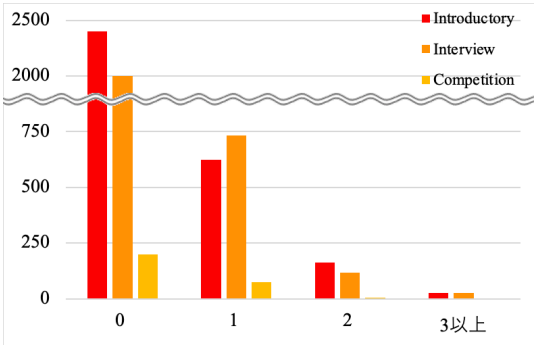

図 4: コードごとの関数内関数の数

また、Multi-modularizing CodeChain は 1 回目のサイクルにおいて Public test を通過するケースがない問題には再生成による改善を行うことができず精度は向上せず、効果を発揮できる問題が限定的である。GPT-3.5 を用いた際であっても半分程度の問題に対しては Public test を一切通過できず、以降のサイクルが機能していない。より根本的な精度向上には Public test を通過しない出力へ作用する LLM 生成コードのデバッグ修正機構が必要になる。

5 おわりに

本研究では LLM によるコード生成タスクを関数を用いて細分化する CodeChain において、細分化を多段化するため関数内関数を用いた構造へ拡張した。結果、精度向上が一部タスクでは確認できたものの精度低下も多く確認された。また、多段階にタスクを分割できた問題は一部に限られる。サイクルを通しての精度の安定化、関数内関数のより効率的な利用方法の模索が今後の課題である。

参考文献

[1] Ashish Vaswani, Noam M. Shazeer, and et al. Attention is all you need. In **Neural Information Processing Systems**, 2017.

[2] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[3] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In **Proceedings of naacL-HLT**, Vol. 1, p. 2. Minneapolis, Minnesota, 2019.

[4] Hugo Touvron, Thibaut Lavril, and et al. LLaMA: Open and Efficient Foundation Language Models. **arXiv e-prints**, p. arXiv:2302.13971, February 2023.

[5] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, ..., and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. **Journal of Machine Learning Research**, Vol. 24, No. 240, pp. 1–113, 2023.

[6] Colin Raffel, Noam Shazeer, and et al. Exploring the limits of transfer learning with a unified text-to-text transformer. **Journal of Machine Learning Research**, Vol. 21, No. 140, pp. 1–67, 2020.

[7] Raymond Li, Loubna Ben allal, and et al. StarCoder: may the source be with you! **Transactions on Machine Learning Research**, 2023. Reproducibility Certification.

[8] Ziyang Luo, Can Xu, and et al. Wizardcoder: Empowering code large language models with evol-instruct. In **The Twelfth International Conference on Learning Representations**, 2024.

[9] Mark Chen, Jerry Tworek, and et al. Jun. Evaluating Large Language Models Trained on Code. **arXiv e-prints**, p. arXiv:2107.03374, July 2021.

[10] Jacob Austin, Augustus Odena, and et al. Program Synthesis with Large Language Models. **arXiv e-prints**, p. arXiv:2108.07732, August 2021.

[11] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In **The Twelfth International Conference on Learning Representations**, 2024.

[12] Dan Hendrycks, Steven Basart, and et al. Measuring coding challenge competence with APPS. In **Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)**, 2021.

[13] Junjie Ye, Xuanting Chen, and et al. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models, 2023.

[14] Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In **The Thirty-eighth Annual Conference on Neural Information Processing Systems**, 2024.

[15] Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. CodeTree: Agent-guided Tree Search for Code Generation with Large Language Models, 2024.

[16] Mirac Suzgun, Nathan Scales, and et al. Challenging big-bench tasks and whether chain-of-thought can solve them. In **ACL (Findings)**, pp. 13003–13051, 2023.

[17] Tom Brown, Benjamin Mann, and et al. Language models are few-shot learners. **Advances in neural information processing systems**, Vol. 33, pp. 1877–1901, 2020.

[18] Zhangyin Feng, Daya Guo, and et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, **Findings of the Association for Computational Linguistics: EMNLP 2020**, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics.

[19] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, **Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing**, pp. 1069–1088, Singapore, December 2023. Association for Computational Linguistics.

[20] Woosuk Kwon, Zhuohan Li, and et al. Efficient memory management for large language model serving with page-dattention. In **Proceedings of the 29th Symposium on Operating Systems Principles**, pp. 611–626, 2023.

[21] Yujia Li, David Choi, and et al. Competition-level code generation with AlphaCode. **Science**, Vol. 378, No. 6624, pp. 1092–1097, December 2022.

A 実験に用いたデータの詳細

ソースコード 5: One-shot で与える出力例

````
STEP 1: GENERATE MODULES:
```module
def count_start_end_chars(words):
    """
    Description: This function counts the number of
        words that start and end with each character.
    Input:
    words (list): A list of binary words.
    Output:
    start_count (defaultdict): A dictionary containing
        the count of words that start with each
        character.
    end_count (defaultdict): A dictionary containing
        the count of words that end with each
        character.
    """
```

```module
def reverse_words(words, reversed_indices):
    """
    Description: This function reverses the specified
        words in the given list.
    Input:
    words (list): A list of binary words.
    reversed_indices (list): A list of indices
        indicating the words to be reversed.
    Output:
    reversed_words (list): A new list of words with the
         specified words reversed.
    """
```

```module
def solve_task(words):
    """
    Description: This function counts the total
        reversed words and their indexes.
    Input:
    words (list): A list of binary words.
    Output:
    total_reversed(int): A integer of the number of
        reversed words.
    reversed_words(list): A list of indexes of reversd
        words.
    """
```

STEP 2: GENERATE SUB-MODULES:

```submodule
    def reverse_word(word):
        """
        Description: This function reverses the given
            word.
        Input:
        word (str): A binary word.
        Output:
        str: The reversed word.
        """
```

```submodule
    def is_reversed(s):
        """
        Description: This function checks if the given
            string is a reversed string.
        Input:
        s (str): A string of binary digits.
        Output:
        bool: True if the given string is a reversed
            string, False otherwise.
        """
```

```submodule
    def count_reversed_words(words):
        """
        Description: This function counts the total
            reversed words and their indexes.
        Input:
        words (list): A list of binary words.
        Output:
        int: The number of reversed words.
        list: A list of indexes of reversd words.
        """
```

STEP 3: GENERATE PYTHON CODE
```python
def count_start_end_chars(words):
    start_chars = {}
    end_chars = {}

    for word in words:
        if word[0] not in start_chars:
            start_chars[word[0]] = 0
        start_chars[word[0]] += 1

        if word[-1] not in end_chars:
            end_chars[word[-1]] = 0
        end_chars[word[-1]] += 1

    return start_chars, end_chars

def reverse_words(words, reversed_indices):

    def reverse_word(word):
        return word[::-1]

    new_words = words.copy()
    for index in reversed_indices:
        new_words[index] = reverse_word(words[index])
    return new_words



def solve_task(words):

    def is_reversed(s):
        return s == s[::-1]

    def count_reversed_words(words):
        count = 0
        indexes = []
        for i in range(len(words)):
            if is_reversed(words[i]):
                count += 1
                indexes.append(i)
        return count, indexes

    count, indexes = count_reversed_words(words)
    print(count)
    print(indexes)



t = int(input())
for _ in range(t):
    n = int(input())
    words = []
    for _ in range(n):
        words.append(input())
    total_reversed, reversed_words = solve_task(words)
    print(total_reversed)
    if total_reversed != 0:
        print(*reversed_words)

```
````