DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines

Anonymous Authors[1]

Abstract

Prompting language models (LMs) have emerged as an integral component within modern natural language processing. However, the probabilistic nature of LMs often leads to the generation of outputs that fail to adhere to domain-specific requirements. Traditional prompting techniques, aimed to guide LMs, face challenges in scalability for inclusion in complex prompting pipelines or require extensive manual effort in developing verbose "prompt templates" to tackle intricate real-world applications. Addressing this, we introduce *Computational Constraints* within the DSPy framework, a state-of-the-art abstraction for building declarative LM pipelines, to revolutionize how we prompt LMs to adhere to user-defined specifications. These constraints are categorized into *Assertions*, which enforce critical conditions leading to LM call interruption upon violation, and *Suggestions*, which offer a more flexible path to output enhancement through self-refinement. We conduct our evaluation across two case studies of question-answering tasks: 1) a multi-hop QA task that involves iterative information retrieval and step-by-step logical reasoning to answer complex questions and 2) an advanced LongForm multi-hop QA task for generating paragraphs with citations to answer questions. Through applying computational constraints, we aim to not only improve compliance to imposed rules and guidelines but also to enhance broader downstream task performance, seeing best gains of 20% and 14.4% respectively.

1. Introduction

Language models (LMs) have become a cornerstone of modern natural language processing, powering a wide range of applications from conversational agents to content generation. However, the inherently probabilistic nature of LMs often results in outputs that may not align with specific user requirements or domain constraints. To address this, researchers have explored various prompting and guidance techniques such as constrained decoding (Hokamp & Liu, 2017; Hu et al., 2019), reflexion (Shinn et al., 2023), and tree-of-thought generation (Yao et al., 2023) to steer LMs towards more desirable outputs. In addition, most techniques for steering LM outputs are explored in isolation, but real-world applications require multi-stage pipelines and agents that decompose complex tasks into manageable calls to LMs and other tools (Qi et al., 2019; Karpas et al., 2022; Khattab et al., 2021; 2023; Chen et al., 2022).

Recently, several language model application frameworks like LangChain (Chase, 2022), LMQL (Beurer-Kellner et al., 2023), and DSPy (Khattab et al., 2023) have been proposed, providing developers simpler interfaces to build complex LM pipelines. Some of these frameworks also provide abstractions over the above-mentioned guidance techniques.

Despite these efforts, the dynamic and complex nature of real-world applications demands a more systematic and scalable approach to guide LMs. Manual prompt engineering or ad-hoc guidance strategies are labor-intensive and lack the flexibility and robustness required for diverse and evolving tasks. Notably, existing solutions lack the expressiveness to *specify arbitrary constraints* on LM pipelines and enable them to *introspectively self-refine outputs*.

To bridge this gap, we introduce ***Computational Constraints***, a novel programming construct designed to enforce user-specified properties on LM outputs within a pipeline framework. Drawing inspiration from runtime assertions and program specifications in traditional programming, computational constraints are expressed as boolean conditions that reflect the desired characteristics of LM outputs. These constraints serve a dual purpose: they act as a mechanism for automated error prevention and a tool for self-refinement, allowing LMs to backtrack and correct their course when

[1] Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

outputs deviate from the specified constraints.

We distinguish between two types of computational constraints: *Assertions* and *Suggestions*, denoted by `Assert` and `Suggest`, respectively. Assertions represent critical conditions that, when violated, cause the LM application to halt, signaling a non-negotiable breach of requirements. Suggestions, on the other hand, denote desirable but non-essential properties; their violation triggers a self-refinement process where the LM attempts to generate more appropriate responses through backtracking and informed re-generation.

We implement our work by extending the capabilities of the DSPy framework, a state-of-the-art framework for building declarative LM pipelines, by integrating computational constraints as a module. This integration not only enables DSPy programs to self-refine and produce outputs that adhere to specific guidelines but also simplifies the debugging experience, providing developers with a clearer understanding of LM behavior within complex pipelines.

We evaluate our approach on the downstream multi-hop QA task of generating brief 1-5 word answers for questions within the HotPotQA dataset. We apply computational constraints to refine search queries by imposing requirements of conciseness and distinctness from past queries to ensure precise information retrieval and answer correctness. Our experiments reveal that these constraints notably guide LMs to enhanced retrieval recall and more accurate responses.

We build on this with a novel enhancement to the task: generating citation-inclusive paragraphs to answer the questions. This task exemplifies the utility of computational constraints, requiring the LM to adhere to strict formatting rules and maintain fidelity to cited sources. By incorporating computational constraints into the DSPy framework, we demonstrate significant improvements in the quality of generated content, showcasing the potential of computational constraints to elevate the performance of LMs in tasks that demand precision and adherence to domain-specific rules.

2. Background and Motivation

While the idea and goals of computational constraints generalize to any LM application programming framework, we implement our work as extensions to the state-of-the-art DSPy (Khattab et al., 2023) framework due to its modular paradigm, flexibility, and extensibility. Consequently, we briefly describe the DSPy programming model for building declarative LM pipelines. Then, we sketch a realistic, motivating example for computation constraints and show their usefulness for self-refinement in LM pipelines.

2.1. The DSPy Programming model

DSPy is a framework for solving advanced tasks with language and retrieval models in a programmatic fashion, unifying techniques for prompting, reasoning, and fine-tuning through composing and declaring modules. The overarching goal of DSPy is to replace brittle "prompt engineering" tricks with composable modules and optimizers.

First, instead of free-form string prompts, DSPy programs use *signatures* to declaratively specify what an LM needs to do. For instance, consume a question and return an answer, as shown below:

```python
qa = dspy.Predict("question -> answer")
qa(question="Where is the Eiffel tower?")
# Output: The Eiffel Tower is located in Paris, France.
```

To use a signature, one declares a *module* with that signature, like we instantiated a `Predict` module above. The core module for working with signatures in DSPy is `Predict`; it stores the supplied signature, formats a prompt to implement the signature, and calls an LM with a list of demonstrations (if any) for prompting.

DSPy modules usually call `dspy.Predict` one or more times and can be defined to translate prompting techniques into modular functions that support any signature, contrasting with the approach of handwriting task-specific prompts with few-shot examples. Consider, for example, the below DSPy module, which implements the popular "chain-of-thought" prompting technique (Wei et al., 2022).

```python
class ChainOfThought(dspy.Module):
    def __init__(self, signature):
        rationale_field = dspy.OutputField(prefix="Reasoning: Let's think step by
       step.")
        signature = dspy.Signature(signature).prepend_output_field(rationale_field)
        self.predict = dspy.Predict(signature)

    def forward(self, **kwargs):
        return self.predict(**kwargs)
```

DSPy modules can be composed in arbitrary pipelines by first declaring the modules needed at initialization and then expressing the pipeline with arbitrary code that calls the modules in a forward method (as shown in the `MultiHopQA` program in Section 2.2). Finally, DSPy provides *teleprompters*, which are optimizers that automate generating good quality demonstrations (few-shot examples) for a task given a metric to optimize.

Challenges. While DSPy signatures provide type hints to LM calls, the framework lacks expressive constructs that developers can use to specify arbitrary constraints the pipeline must satisfy. Additionally, one can imagine the LM pipeline using these constraints to refine its outputs.

To address these challenges, we integrate computational constraints as first-class primitives in DSPy. In the style of Pythonic assertions, they are intuitive constructs that allow

DSPy to provide guidelines for common failures, facilitate backtracking and self-correction of LM calls, and debug pipeline failures. In what follows, we describe a motivating example of a DSPy program that uses computational constraints for multi-hop question answering.

2.2. Motivating Example

Alice is a developer building an LM pipeline for multi-hop question-answering. The task involves the LM performing a series of inferential steps (multi-hop) before answering a question while utilizing a retriever to retrieve relevant context.

In a simple implementation of this pipeline in DSPy, Alice may design the pipeline below where the LM generates search queries to collect relevant context and aggregate them to generate the answer. This simulates the information flow in popular multi-hop question-answering works such as Baleen (Khattab et al., 2021) and IRCoT (Trivedi et al., 2022).

```python
class MultiHopQA(dspy.Module):
    def __init__(self):
        self.retrieve = dspy.Retrieve(k=3)
        self.gen_query = dspy.ChainOfThought("context, question -> search_query")
        self.gen_answer = dspy.ChainOfThought("context, question -> answer"))

    def forward(self, question):
        context = []

        for hop in range(2):
            query = self.gen_query(context=context, question=question).search_query
            context += self.retrieve(query).passages

        return self.gen_answer(context=context, question=question)
```

However, certain issues with the pipeline might affect its performance. For instance, since questions are complex, the generated search query could be long and imprecise, resulting in irrelevant retrieved context. Another issue is that similar multi-hop queries would result in redundant retrieved context. One might observe that these are properties of generated queries that are *computationally checkable* and, if expressible as *constraints* on the pipeline, might improve its performance.

Figure 1 shows a DSPy program with computational constraints for this task. To mitigate the issues above, Alice introduces two soft computational constraints: first, she restricts the length of the query to be less than 100 characters, aiming for precise information retrieval. Second, she requires the query generated at each hop to be dissimilar from previous hops, discouraging retrieval of redundant information. She specifies these as *soft constraints*, using the `Suggest` construct, allowing the pipeline to backtrack to the failing module and retry on failure. The LM is made aware of its past attempts and suggestions on retrying, enabling constraint-guided self-refinement.

In Section 5.3, we evaluate this pipeline on the HotPotQA (Yang et al., 2018) dataset. We find that enabling the developer to express two simple suggestions improves the retriever's recall (by 2-5%) and the accuracy of generated answers (by 1%).

3. Semantics of Computational Constraints

To help with the goals mentioned above, in this work, we introduce ***Computational Constraints***. We define computational constraints as programmatic elements that dictate certain conditions or rules that must be adhered to during the execution of an LM pipeline. These constraints are designed to ensure that the pipeline's behavior aligns with specified invariants or guidelines, enhancing the reliability, predictability, and correctness of the pipeline's output.

We categorize computational constraints into two well-defined programming constructs, namely ***Assertions*** and ***Suggestions***. They are constructs that enforce constraints and provide guidance within an LM pipeline's execution flow. Assertions enable developers to enforce strict invariants, acting as critical checkpoints that the pipeline must satisfy to continue execution. Suppose an assertion's condition evaluates to false. In that case, it triggers an immediate transition to an error state, resulting in the termination of the pipeline and the raising of an `AssertionError` with an accompanying message. In contrast, suggestions are softer constraints that recommend but do not mandate conditions that may guide the LM pipeline toward desired domain-specific outcomes. When a suggestion's condition is not met, the pipeline enters an alternate execution path to log a warning, raise a `SuggestionError` that can be caught or handled, or invoke a recovery mechanism. This allows the pipeline to potentially adjust its behavior in response to the suggestion, thereby providing a more flexible and resilient execution model that can recover from suboptimal states.

In the following sections, we define the semantics of these constructs more formally.

3.1. `Assert`

The `Assert` construct enforces invariants within the LM pipeline. The semantics of an assertion can be defined in terms of a state transition system where σ represents the pipeline's state:

$$\begin{aligned}
\sigma \vdash \texttt{Assert}(e, m) \rightarrow \sigma' \quad & \text{if eval}(\sigma, e) = \text{true} \\
\sigma \vdash \texttt{Assert}(e, m) \rightarrow \bot \quad & \text{if eval}(\sigma, e) = \text{false}
\end{aligned}$$

Here, $\text{eval}(\sigma, e)$ denotes the evaluation of expression e in state σ. If e evaluates to true, the pipeline transitions to a new state σ' that is identical to σ. If e evaluates to false, the pipeline transitions to an error state $\bot$, and an `AssertionError` with message m is raised, halting the exe-

```python
class MultiHopQAWithAssertions(dspy.Module):
    def __init__(self):
        self.retrieve = dspy.Retrieve(k=3)
        self.generate_query = [dspy.ChainOfThought("context, question -> search_query") for _ in range(2)]
        self.generate_answer = dspy.ChainOfThought("context, question -> answer"))

    def forward(self, question):
        context, queries = [], [question]

        for hop in range(2):
            queries_str = "; ".join(queries)
            query = self.generate_query[hop](context=context, question=question).search_query

            dspy.Suggest(len(query) <= 100, "Query should be short and less than 100 characters")

            dspy.Suggest(is_query_distinct(query, queries), f"Query should be distinct from {queries_str}"))

            queries.append(query)
            context += self.retrieve(query).passages

        pred = self.generate_answer(context=context, question=question)
        return dspy.Prediction(context=context, answer=pred.answer)
```

Figure 1. DSPy program with computational constraints for multi-hop question answering task with a retriever. We introduce two suggestions: (1) the query to retriever should be shorter than 100 characters; (2) the query to retriever should differ from previous queries.

cution.

3.2. `Suggest`

The `Suggest` construct provides non-binding guidance to the LM pipeline. Its semantics can be defined as follows:

$$\sigma \vdash \texttt{Suggest}(e, m) \rightarrow \sigma' \quad \text{if eval}(\sigma, e) = \text{true}$$
$$\sigma \vdash \texttt{Suggest}(e, m) \rightarrow \sigma'' \quad \text{if eval}(\sigma, e) = \text{false}$$

Here, if the expression e evaluates to true, the pipeline transitions to a new state σ' that is identical to σ. If e evaluates to false, the pipeline transitions to an alternative state σ'' where it may attempt to recover or adjust its behavior based on the suggestion. The transition to σ'' may involve logging the message m, raising a `SuggestionError` that can be caught and handled, or triggering a backtracking mechanism to retry a previous computation with new information.

Handling `SuggestionError` is an important aspect of the semantics of suggestions. The LM pipeline may define a handler H that determines the response to a failed suggestion:

$$\sigma \vdash \texttt{Suggest}(e, m) \rightarrow H(\sigma, m) \quad \text{if eval}(\sigma, e) = \text{false}$$

Here, the handler H takes the current erring state σ and the suggestion message m as inputs and returns a new state σ''. The handler may involve complex logic, such as backtracking, retrying with additional information, or altering the pipelines's control flow. We describe these details in the next section.

4. Implementation

We introduce the proposed computational constraints as plug-in interfaces in the DSPy framework according to the semantics in Section 3. Next, we describe details about the design of our APIs and how we implement the semantics of both `Assert` and `Suggest` in DSPy.

4.1. API Design

```python
dspy.Assert(constraint: bool, message: Optional[str])
dspy.Suggest(constraint: bool, message: Optional[str],
                backtrack: Optional[module])
```

We inherit a simple API design for computational constraints. Both suggestions and assertions will take a boolean value `constraint` as input. Note that the computation for constraint can contain other DSPy modules, which potentially invoke other language models to compute the result of the constraint. Then, the user provides an optional error message, which is used for error logging and feedback construction for backtracking and refinement. Finally, the suggestion API `dspy.Suggest` contains an additional optional `backtrack` argument, which points to the target module to backtrack to if the constraint fails.

```python
class LongFormQAWithAssertions(dspy.Module):
    def __init__(self, passages_per_hop=3):
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_query = [dspy.ChainOfThought("context, question -> search_query") for _ in range(2)]
        self.generate_cited_paragraph = dspy.ChainOfThought("context, question -> paragraph")

    def forward(self, question):
        context = []
        for hop in range(2):
            query = self.generate_query[hop](context=context, question=question).query
            context += self.retrieve(query).passages

        pred = self.generate_cited_paragraph(context=context, question=question)
        pred = dspy.Prediction(context=context, paragraph=pred.paragraph)

        dspy.Suggest(citations_check(pred.paragraph), f"Make sure every 1-2 sentences has citations. If any 1-2 sentences lack citations, add them in ’text... [x].’ format.", backtrack=GenerateCitedParagraph)

        citation_faithfulness_score, unfaithful_outputs = citation_faithfulness(pred)

        if unfaithful_outputs:
            unfaithful_pairs = [(output[’text’], output[’context’]) for output in unfaithful_outputs]
            for unfaithful_text, context in unfaithful_pairs:
                dspy.Suggest(check_unfaithful(unfaithful_pairs), f"Make sure your output is based on the following context: ’{context}’.", backtrack=GenerateCitedParagraph)
        else:
            return pred
        return pred
```

Figure 2. DSPy program with computational constraints for long-form paragraph multi-hop question answering task with a retriever. We introduce two suggestions: (1) asserting every 1-2 sentences has a citation; (2) every text segment preceding a citation is faithful to its cited reference.

4.2. Error Handlers

To implement various strategies of both assertions and suggestions for different use cases, we exploit Python's native error and exception handling.

We first encode error-handling logic as function wrappers and provide a DSPy primitive `constraint_tranform` to wrap any DSPy modules with handlers. As stated in Section 3, when the constraints in `Assert` and `Suggest` are false, they raise `AssertionError` and `SuggestionError`, respectively. Then, the dedicated error handling clause in the function wrapper can reroute the errors to the correct semantics. For example, in the default case, an `AssertionError` will be raised to abort the program marking a violation of a hard constraint from the user. `SuggestionError`, however, would be handled by backtracking to the previous module and retrying with an appended error message and previous erring output, allowing the pipeline to self-refine.

4.3. Backtracking

To implement backtracking in DSPy, we introduce a new auxiliary module called `Retry`. This module acts as a lightweight wrapper for any DSPy module, providing additional information about all previously unsuccessful predictions. As a result, when DSPy determines the need to backtrack to a specific module, it calls `Retry` which leverages both the failed predictions and their corresponding error messages from assertions or suggestions. This dynamic signature modification prompts the original module to refine its outputs while being self-aware and informed of its past attempts. Consequently empowering the language model to make more informed decisions in subsequent iterations.

5. Evaluation

While there are several axes to evaluate programming constructs such as assertions, here, we focus on investigating the following hypotheses:

H1 Computational constraints facilitate general-purpose self-correction and refinement for LM pipelines by showing past outputs and error messages to the LM.

H2 Computational constraints-guided self-refinement can also enable LM pipelines to improve downstream application performance.

We will explore these hypotheses using two related question-answering (QA) tasks and their corresponding DSPy programs. First, we explore the popular multi-hop QA task and evaluate the effectiveness of computational constraints to self-refine search queries for better retrieval recall as well as overall answer correctness. Next, we introduce a novel enhancement to the standard multi-hop QA task: LongForm QA. Here, the pipeline is expected to produce a long-form answer that includes citations to referenced retrieved context information. The objective of this task is to use computational constraints to not just answer a question accurately but to generate a cohesive, well-structured paragraph that is faithful to the retrieved context.

5.1. Metrics

For evaluating each task, we consider two distinct categories of metrics:

- **Intrinsic Quality** measures the degree to which the outputs conform to the computational constraints specified within the program. This metric serves as a benchmark for the system's ability to pass validation checks that confirm it is functioning as intended. It reflects the internal consistency and adherence to the constraints, which are critical for ensuring the reliability of the LM pipeline's self-correction and refinement processes.
- **Extrinsic Quality** assesses the impact of computational constraints on the LM pipeline's performance in broader, task-specific contexts. Here, constraints act as guiding principles that indirectly influence the system's effectiveness in achieving its overall objectives. By serving as proxies for more complex goals, the constraints provide insights into how their integration can lead to enhancements in downstream application performance, such as improved accuracy in question-answering tasks or the generation of more coherent and contextually accurate long-form answers.

These metrics will enable us to systematically investigate the hypotheses that computational constraints can facilitate self-correction and refinement in LM pipelines and that such guided self-refinement can enhance the performance of downstream applications.

5.2. Dataset & Models

For both tasks, we utilize the HotPotQA dataset in the open-domain "fullwiki" setting for the evaluation of our downstream tasks. Since the HotPotQA test set is not publicly accessible, we reserve the official validation set for testing and sample 300 examples for that. We then partition the official training set into subsets: 70% for training and 30% for validation. We only focus on examples labeled as "hard" within the dataset to align with the criteria marked by the official validation and test sets. Finally, for training and development sets, we sample 300 examples each.

Table 1. Evaluation of MultiHop Question Answering with HotPotQA. This table presents comparisons across various strategies, including `ZeroShot` (uncompiled) and `FewShot` (compiled with DSPy's `BootstrapFewShotWithRandomSearch` teleprompter), both with and without assertions (`NoSuggest` and `Suggest`). Metrics measured are Retrieval Score and Correctness (Section 5.3.1)

Configuration	**Ret. Score**	**Correctness**
`ZeroShot–NoSuggest`	34.67%	45.67%
`ZeroShot–Suggest`	39.33%	46.33%
`FewShot–NoSuggest`	40.33%	49.33%
`FewShot–Suggest`	42.00%	50.00%

For retrieval, we make use of a search corpus which we employ through the official Wikipedia 2017 "abstracts" dump of HotPotQA using a ColBERTv2 (Santhanam et al., 2021) retriever. We test the program on the OpenAI's `gpt-3.5-turbo` (Brown et al., 2020) language model with `max_tokens=500` and `temperature=0.7` for experimental reproducibility.

5.3. Case Study: Multi-Hop QA

5.3.1. Task & Metrics

In this task, we explore complex question answering through a multi-hop prompting technique. This task involves generating queries related to the given question, iteratively retrieving relevant context from a search index, and synthesizing this information to generate an answer.

Figure 1 shows an implementation of this task in DSPy. As shown, in each hop, a `dspy.ChainOfThought` module generates a search query for the retriever based on the current accumulated context and the initial question. A `dspy.Retrieve` then fetches k relevant passages which are aggregated to the context. Once the loop reaches the maximum number of hops, a `dspy.ChainOfThought` module produces the final answer.

With this task and LM pipeline, we aim to produce a robust and thorough reasoning process leading to accurate answers to questions.

We assess performance using two metrics: the Retrieval Score (intrinsic), which quantifies the proportion of correctly retrieved `gold` passages, and the Answer Correctness (extrinsic), which checks if the generated answer precisely matches the `gold` standard answer.

5.3.2. Constraints Specified

To evaluate computational constraints on this task, we introduce two simple suggestions. First, we suggest that the query generated be short and less than 100 characters, with

Table 2. Evaluation of LongFormQA with Citations Task. This table covers a comparative analysis across different strategies, including ZeroShot (uncompiled) and FewShot (includes few-shot example demonstrations, compiled with DSPy's BootstrapFewShotWithRandomSearch teleprompter), both with and without assertions (NoSuggest and Suggest). Metrics measured are Citation Faithfulness, Recall, Precision, and Answer Correctness (Section 5.4.1)

Configuration	Citation Faithfulness	Recall	Precision	Answer Correctness
ZeroShot–NoSuggest	76.00%	51.83%	59.38%	66.67%
ZeroShot–Suggest	88.67%	56.33%	64.83%	67.33%
FewShot–NoSuggest	86.33%	62.83%	75.28%	69.33%
FewShot–Suggest	96.00%	62.50%	73.78%	69.33%

the goal of reducing imprecise retrieval. Second, we require that a generated query is distinct from previous queries. This discourages the pipeline from retrieving redundant context in each hop. Note that the condition checks for both constraints are implemented using simple Python code. Overall, these constraints enable the self-refinement of the search queries generated by the program.

5.3.3. Evaluation

For this task, we evaluate the MultiHopQA program under various configurations to understand the impact of computational constraints and few-shot learning techniques. Table 1 shows the results of this evaluation.

Our simplest baseline is the MultiHopQA in a zero-shot, uncompiled configuration (ZeroShot–NoSuggest). When we include the mentioned assertions within this zero-shot configuration (ZeroShot–Suggest), we see a notable improvement of ≈5% in the retrieval score. We also observe an indirect ≈1% increase in answer correctness due to refined, concise, and distinct query generation.

We also take advantage of optimizations provided by DSPy compiling frameworks, utilizing the BootstrapFewShotWithRandomSearch 'teleprompter' which automatically generates and integrates few-shot examples doing a random search over a set of candidates within the MultiHopQA program. We include a maximum of 2 bootstrapped few-shot examples to fit within the context window and compile 6 candidates on the devset examples. We apply this evaluation approach to both the vanilla MultiHopQA and the MultiHopQA with Assertions programs. In the non-assertion configuration (FewShot–NoSuggest), we see a strong performance gain in retrieval score and correctness, which highlights the efficacy of utilizing few-shot techniques to refine the querying and reasoning processes. We find that combining few-shot learning with assertions (FewShot–Suggest) further demonstrates improvements of ≈1-2% on both metrics.

5.4. Case Study: LongForm QA

5.4.1. Task & Metrics

In this task, we build on the multi-hop QA (Section 5.3) task by expecting long-form answers to questions that include citations to referenced context.

Figure 2 shows an implementation of this task in DSPy. As shown, it is nearly identical to Figure 1 outside of the introduction of a new dspy.ChainOfThought module that generates cited paragraphs referencing the retrieved context.

With this task and LM pipeline, we aim not just to produce accurate answers but to generate well-structured long-form answers that are faithful to the retrieved context.

We assess intrinsic performance using a novel metric–Citation Faithfulness–where an LM checks if the text preceding a citation accurately reflects the cited context, outputting a boolean for faithfulness. As extrinsic metrics, we use: (1) Answer Correctness, verifying if the gold answer is correctly incorporated; (2) Citation Precision, gauging the proportion of correctly cited titles; and (3) Citation Recall, measuring the coverage of gold titles cited.

5.4.2. Constraints Specified

We introduce more complex suggestions here, unlike in Section 5.3. First, we suggest that every pair of sentences generated has citations in an intended format. This is checked by a simple Python function citations_check. Next, we specify that the text preceding any citation must be faithful to the cited context, ensuring that the reference text accurately represents the content of the cited information. Since this is a fuzzy condition, we employ an LM to perform this check. Notably, the robust API design of Suggest allows the user to specify arbitrary expressions as conditional checks, such as an LM call.

5.4.3. Evaluation

We extend our analysis to the LongFormQA task to understand how different configurations impact the quality of generated paragraphs with citations to answer questions in terms of citation faithfulness, recall, precision, and answer correctness. We employ a similar evaluation methodology used in the MultiHopQA evaluation (5.3.3).

We start with the simplest baseline: LongFormQA in a zero-shot, uncompiled configuration (`ZeroShot–NoSuggest`). Introducing assertions in this zero-shot setting (`ZeroShot–Suggest`) yields a highly significant ≈12% improvement on the intrinsic citation faithfulness metric, indicating the effect of imposed constraints in maintaining proper citation formatting and referencing. Furthermore, we also observe gains across all extrinsic metrics–recall, precision, and correctness–reflecting the value of effective citation inclusion in enhancing the overall quality of the generated paragraphs in providing composite answers.

Again, we explore enhancements by compiling with few-shot examples in DSPy using a random search (Section 5.3.3). In the non-assertion configuration (`FewShot–NoSuggest`), we observe a performance improvement across metrics as the few-shot demonstrations further refine information retrieval and citations. When including assertions (`FewShot–Suggest`), we continue to see notable gains of ≈10% in citation faithfulness. However, we do see a decline in recall and precision, reflecting some complex interplay between adherence to computational constraints and optimizing few-shot examples for answer correctness. Yet we observe that introducing computational constraints does not decrease performance on the fundamental downstream metric of answer correctness.

6. Related Work

6.1. Self-refinement and correction

Our contributions address a key limitation of expressing computational constraints (Zhu et al., 2023) in existing language model programming frameworks such as DSPy. The ambiguity of user prompts highlights the significance of clear task decomposition which can be supported by tools within LLM frameworks (Liu et al., 2023). By integrating Python-style assertions, we ensure programmers have the capability to clearly express computational constraints on DSPy programs and assert effective program behavior.

Self-refinement of LLMs (Shridhar et al., 2023) is central to this approach in making DSPy autonomous and context-aware (Tyen et al., 2023). Enforcing methodologies of iterative refinement using error feedback (Xu et al., 2023) and utilizing reasoning capabilities through presenting past generations and feedback for correction (Qiu et al., 2023) resonates with the objective of DSPy assertions.

6.2. Programming with constraints

Programming with constraints is common in most programming languages. Popular programming languages like Java (Bartetzko et al., 2001) or Python (Python Software Foundation, 2023) all support expressing assertions as first-class statements to perform runtime checks of certain properties. The `Assert` construct we introduce is equivalent to these traditional assertions in popular programming languages.

However, most runtime checks can only be used to warn the programmer or abort the execution early. Particularly, with language models being non-deterministic and optimizable with better prompts, we introduce a more powerful computational constraint called suggestions: `Suggest`. Different from the traditional runtime checks, `Suggest` is able to backtrack the DSPy program to provide additional information and feedback in the prompt that guides the LM program to make better predictions in order to eventually pass the constraint.

6.3. Program Synthesis and Sketching

Classical program synthesis generates programs based on a given specification, involving a search for candidate programs that match the specification's structure (Pnueli & Rosner, 1989; Gulwani et al., 2017). In this space, particularly relevant to DSPy's computational constraints are approaches such as Sketching (Solar-Lezama, 2008) and Template-based synthesis (Srivastava et al., 2013). In sketching (Solar-Lezama, 2009), the programmer specifies a high-level structure ("sketch") with placeholders ("holes") for the synthesis engine to complete. The sketch includes assertions, and the correctness requirement is that these assertions hold for all inputs within the bound specified by the synthesizer. We note that the specification assertions in Sketching correspond to the computational constraints (`Assert` and `Suggest`) we propose, while a synthesizer parallels the LM pipeline framework, like the DSPy compiler, striving to compile an optimized prompt.

6.4. Factual Consistency and Citations

In this work, we evaluate DSPy computation constraints on the task of long-form paragraph generation with citations to answer questions (Gao et al., 2023). This task is inspired by several studies with the objective of ensuring factual consistency and citation quality from LLM outputs. Existing frameworks (Min et al., 2023) highlight the complexity of evaluating factual faithfulness in natural language generation. Proposed methodologies like TrueTeacher (Gekhman et al., 2023) and Chain-of-Verification (Dhuliawala et al., 2023) demonstrate LLM-based factual consistency evaluations which align with the assertion-based methodology in DSPy for ensuring text-to-citation faithfulness. Furthermore, benchmarks for automatic attribution evaluation (Yue et al., 2023) and verifying citations for text generations reflect direct applications for the motivation of this paper, focused on showing the efficacy of asserting the presence and respective accuracy of citing references.

7. Conclusion

We explore the integration of assertions within the DSPy framework in this paper, emphasizing the capability to guide LLMs in self-refining outputs for downstream tasks. Our approach allows for the expression of computational constraints which are enforced in a manner inspired by Pythonic-style assertions. By enabling DSPy programs to autonomously backtrack and self-correct, we have opened avenues for dynamic applications with more context-aware LLM generations.

References

Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H. Jass - java with assertions. In Havelund, K. and Rosu, G. (eds.), *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001, Paris, France, July 23, 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pp. 103–117. Elsevier, 2001. doi: 10.1016/S1571-0661(04)00247-6. URL https://doi.org/10.1016/S1571-0661(04)00247-6.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chase, H. LangChain, October 2022. URL https://github.com/langchain-ai/langchain.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., and Weston, J. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023.

Gao, T., Yen, H., Yu, J., and Chen, D. Enabling large language models to generate text with citations. *arXiv preprint arXiv:2305.14627*, 2023.

Gekhman, Z., Herzig, J., Aharoni, R., Elkind, C., and Szpektor, I. Trueteacher: Learning factual consistency evaluation with large language models. *arXiv preprint arXiv:2305.11171*, 2023.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Hokamp, C. and Liu, Q. Lexically constrained decoding for sequence generation using grid beam search. *arXiv preprint arXiv:1704.07138*, 2017.

Hu, J. E., Khayrallah, H., Culkin, R., Xia, P., Chen, T., Post, M., and Van Durme, B. Improved lexically constrained decoding for translation and monolingual rewriting. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 839–850, 2019.

Karpas, E., Abend, O., Belinkov, Y., Lenz, B., Lieber, O., Ratner, N., Shoham, Y., Bata, H., Levine, Y., Leyton-Brown, K., et al. Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*, 2022.

Khattab, O., Potts, C., and Zaharia, M. Baleen: Robust multi-hop reasoning at scale via condensed retrieval. *Advances in Neural Information Processing Systems*, 34: 27670–27682, 2021.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. Dspy: Compiling declarative language model calls into self-improving pipelines. *CoRR*, abs/2310.03714, 2023. doi: 10.48550/ARXIV.2310.03714. URL https://doi.org/10.48550/arXiv.2310.03714.

Liu, Z., Lai, Z., Gao, Z., Cui, E., Zhu, X., Lu, L., Chen, Q., Qiao, Y., Dai, J., and Wang, W. Controlllm: Augment language models with tools by searching on graphs. *arXiv preprint arXiv:2310.17796*, 2023.

Min, S., Krishna, K., Lyu, X., Lewis, M., Yih, W.-t., Koh, P. W., Iyyer, M., Zettlemoyer, L., and Hajishirzi, H. Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*, 2023.

Pnueli, A. and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 179–190, 1989.

Python Software Foundation. 7. simple statements. https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement, 2023. Accessed: 2023-12-01.

Qi, P., Lin, X., Mehr, L., Wang, Z., and Manning, C. D. Answering complex open-domain questions through iterative query generation. *arXiv preprint arXiv:1910.07000*, 2019.

Qiu, L., Jiang, L., Lu, X., Sclar, M., Pyatkin, V., Bhagavatula, C., Wang, B., Kim, Y., Choi, Y., Dziri, N., et al. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. *arXiv preprint arXiv:2310.08559*, 2023.

Santhanam, K., Khattab, O., Saad-Falcon, J., Potts, C., and Zaharia, M. Colbertv2: Effective and efficient retrieval via lightweight late interaction. *arXiv preprint arXiv:2112.01488*, 2021.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K. R., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Shridhar, K., Sinha, K., Cohen, A., Wang, T., Yu, P., Pasunuru, R., Sachan, M., Weston, J., and Celikyilmaz, A. The art of llm refinement: Ask, refine, and trust. *arXiv preprint arXiv:2311.07961*, 2023.

Solar-Lezama, A. *Program synthesis by sketching*. University of California, Berkeley, 2008.

Solar-Lezama, A. The sketching approach to program synthesis. In *Asian symposium on programming languages and systems*, pp. 4–13. Springer, 2009.

Srivastava, S., Gulwani, S., and Foster, J. S. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15:497–518, 2013.

Trivedi, H., Balasubramanian, N., Khot, T., and Sabharwal, A. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*, 2022.

Tyen, G., Mansoor, H., Chen, P., Mak, T., and Cărbune, V. Llms cannot find reasoning errors, but can correct them! *arXiv preprint arXiv:2311.08516*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.

Xu, W., Deutsch, D., Finkelstein, M., Juraska, J., Zhang, B., Liu, Z., Wang, W. Y., Li, L., and Freitag, M. Pinpoint, not criticize: Refining large language models via fine-grained actionable feedback. *arXiv preprint arXiv:2311.09336*, 2023.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

Yue, X., Wang, B., Zhang, K., Chen, Z., Su, Y., and Sun, H. Automatic evaluation of attribution by large language models. *arXiv preprint arXiv:2305.06311*, 2023.

Zhu, Z., Xue, Y., Chen, X., Zhou, D., Tang, J., Schuurmans, D., and Dai, H. Large language models can learn rules. *arXiv preprint arXiv:2310.07064*, 2023.

A. Error handling and transformation code