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

Arnav Singhvi [*1] **Manish Shetty** [*1] **Shangyin Tan** [*1]
Christopher Potts [2] **Koushik Sen** [1] **Matei Zaharia** [1] **Omar Khattab** [2]

Abstract

Chaining language model (LM) calls as composable modules is fueling a powerful way of programming. However, ensuring that LMs adhere to important constraints remains a key challenge, one often addressed with heuristic "prompt engineering". We introduce **LM Assertions**, a new programming construct for expressing computational constraints that LMs should satisfy. We integrate our constructs into the recent DSPy programming model for LMs, and present new strategies that allow DSPy to compile programs with arbitrary LM Assertions into systems that are more reliable and more accurate. In DSPy, LM Assertions can be integrated at compile time, via automatic prompt optimization, and and/or at inference-time, via automatic self-refinement and backtracking. We report on two early case studies for complex question answering (QA), in which the LM program must iteratively retrieve information in multiple hops and synthesize a long-form answer with citations. We find that LM Assertions improve not only compliance with imposed rules and guidelines but also enhance downstream task performance, delivering intrinsic and extrinsic gains up to 35.7% and 13.3%, respectively.

1. Introduction

Language models (LMs) now power various applications in Natural Language Processing (NLP) from conversational agents to writing assistants. However, the probabilistic nature of LMs often results in outputs that may not align with the constraints of the domain or the larger pipeline in which the LM is used. To address this, researchers have explored various techniques, including applying constrained decoding (Hokamp & Liu, 2017; Hu et al., 2019), monitoring model assertions (Kang et al., 2020), exploring approaches for self-reflection and tree search (Madaan et al., 2023; Shinn et al., 2023; Yao et al., 2023), or even building domain-specific languages and systems such as LMQL (Beurer-Kellner et al., 2023) and Guardrails (Rebedea et al., 2023) to steer LMs towards more controllable outputs.

Recently, several LM application frameworks like LangChain (Chase, 2022) and LM programming models like DSPy (Khattab et al., 2022; 2023) have been proposed, providing developers with simpler interfaces to build complex LM pipelines. While these frameworks offer a number of key features to control LM outputs (e.g., DSPy's automatic compiler), they lack the expressiveness to *specify arbitrary computational constraints* on LM pipelines and enable them to learn from these constraints and to introspectively *self-refine outputs*. While some of this may be achieved via pain-staking "prompt engineering" or other ad-hoc guidance strategies, this would be labor-intensive and in general conflates high-level program design with the low-level exploration of teaching LMs how to follow constraints and using them iteratively for self-refinement.

We introduce ***LM Assertions***, 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, LM Assertions are expressed as boolean conditions that reflect the desired characteristics of LM outputs. Besides serving as conventional runtime monitors, LM Assertions create multiple new ways to improve LM programs. First, they can be used to facilitate **runtime self-refinement** in LM pipelines. When an LM Assertion fails, instead of simply terminating, the pipeline backtracks and retries the failing pipeline module. LM assertions provide feedback on retry attempts; they inject erring outputs and error messages to the prompt to introspectively self-refine outputs. Figure 1 illustrates this self-refinement process in a DSPy pipeline. Second, LM Assertions can enable **guided prompt optimizers** at compile time. Integrated with existing automatic prompt optimizers, like in DSPy, they can enable compiling harder few-shot examples, which tunes LM programs into systems that are more robust and

[*]Equal contribution [1]University of California, Berkeley, US [2]Stanford University, Stanford, US.

accurate. In the same vein, the failing LM assertions can also be used to collect counter-examples for optimizers.

We distinguish between two types of LM Assertions: (hard) *Assertions* and (soft) *Suggestions*, denoted by `Assert` and `Suggest`, respectively. Hard assertions represent critical conditions that, when violated after a maximum number of retries, cause the LM pipeline to halt, signaling a non-negotiable breach of requirements. On the other hand, suggestions denote desirable but non-essential properties; their violation triggers the self-refinement process, but exceeding a maximum number of retries does not halt the pipeline. Instead, the pipeline continues to execute the next module.

We implement our work by extending the capabilities of the DSPy framework, a state-of-the-art framework for building and automatically optimizing declarative LM pipelines, by integrating LM Assertions 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. In addition, by combining LM Assertions with prompt optimizers in DSPy, we envision the potential to bootstrap harder few-shot demonstrations to make the pipeline more robust and performant.

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 LM Assertions to refine retrieval queries by imposing requirements of conciseness and distinctness from past queries to ensure precise information retrieval and answer correctness. Our experiments show that by incorporating the assertion, search queries are 27.2%–35.7% more likely to be distinct from previous queries; thus, retrieval queries can fetch more useful information in the context. As a result, the useful passages retrieved increase 4.2%–13.3%, and the final answer correctness increases 1.3%–1.4%.

We build on this with an enhancement to the task: generating citation-inclusive paragraphs to answer the questions (Gao et al., 2023). This task exemplifies the utility of LM Assertions, 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 with more than 16.7% more faithful citations, showcasing the potential of LM Assertions to elevate the performance of LMs in tasks that demand precision and adherence to domain-specific rules.

2. Background and Motivation

The goals of LM Assertions are general and can be applied to any LM program. Due to its modular paradigm, flexibility, and extensibility, we implement our work as extensions to the state-of-the-art DSPy (Khattab et al., 2023) framework. Below, we briefly describe the DSPy programming model for building declarative LM pipelines and *compiling* them into auto-optimized prompt (or finetune) chains. We then sketch a realistic, motivating example for LM Assertions and show their usefulness for self-refinement in LM pipelines.

2.1. The DSPy Programming model

DSPy is a framework for programmatically solving advanced tasks with language and retrieval models through composing and declaring modules. The overarching goal of DSPy is to replace brittle "prompt engineering" tricks with composable modules and automatic (typically discrete) optimizers.

First, instead of free-form string prompts, a programmer using DSPy will define a *signature* to declaratively specify what an LM needs to do. For instance, a module may need to consume a question and return an answer, as shown below:

```
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, the programmer declares a *module* with that signature, like we defined a `Predict` module above. The core module for working with signatures in DSPy is `Predict` . Internally, it stores the supplied signature. When the signature is called, like a function, it constructs a formatted prompt according to the signature's inputs and outputs. Then, it calls an LM with a list of demonstrations (if any) following this format for prompting.

DSPy modules usually call `dspy.Predict` one or more times. They generally encapsulate prompting techniques, turning them into modular functions that support any signature. This contrasts with handwriting task-specific prompts with manually tuned instructions or few-shot examples. Consider, for example, the below DSPy module from Khattab et al. (2023), which implements the popular "chain-of-thought" prompting technique (Wei et al., 2022).

```
class ChainOfThought(dspy.Module):
  def __init__(self, signature):
    rationale_field = dspy.OutputField(prefix="Reasoning: 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 `ChainOfThought` module above and the `MultiHopQA` program in Section 2.2). Finally, DSPy provides *teleprompters*, which are optimizers that can, among other things, automate

generating good quality demonstrations (few-shot examples) or instructions for a task given a metric to optimize. We may also refer to the few-shot example selection process as *compiling* the LM pipeline application.

Challenges. DSPy signatures provide type hints that softly shape LM's behavior. However, the framework currently lacks constructs developers can use to specify arbitrary computational constraints the pipeline *must* satisfy. Additionally, one can imagine the LM pipeline using these constraints to refine its outputs and to teach the LM to respect these specifications at compile time.

To address these challenges, we integrate LM Assertions as first-class primitives in DSPy. In the style of Pythonic assertions, they are intuitive constructs that allow DSPy to constrain LM outputs. They are flexible in that they can be strict restrictions, softer guidelines for backtracking and self-correction of LM calls, or simple debugging statements. In what follows, we describe a motivating example of a DSPy program that uses LM Assertions for multi-hop question answering.

2.2. Motivating Example

Aiden 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 get relevant context.

In a simple DSPy implementation, Aiden may design the pipeline below, where the LM generates search queries to collect relevant context and aggregate them to generate the answer.[1]

```
class MultiHopQA(dspy.Module):
    def __init__(self):
        self.retrieve = dspy.Retrieve(k=3)
        self.gen_query = dspy.ChainOfThought("context, question -> 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).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 LM Assertions for this task. To mitigate the issues above, Aiden introduces two soft LM Assertions: first, they restrict the length of the query to be less than 100 characters, aiming for precise information retrieval. Second, they require the query generated at each hop to be dissimilar from previous hops, discouraging retrieval of redundant information. They specify these as *soft constraints* using the `Suggest` construct. The force of this construct is to allow the pipeline to backtrack to the failing module and try again. On retrying, the LM prompt also contains its past attempts and suggestion messages, 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 4.2%–13%) and the accuracy of generated answers (by 1.3%–1.4%).

3. Semantics of LM Assertions

To help with the goals mentioned above, in this work, we introduce ***LM Assertions*** and integrate them in DSPy. We define LM Assertions as programmatic elements that dictate certain conditions or rules that must be adhered to during the execution of an LM pipeline. These constraints 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 LM Assertions into two well-defined programming constructs, namely ***Assertions*** and ***Suggestions***, denoted by the constructs `Assert` and `Suggest`. They are constructs that enforce constraints and guide an LM pipeline's execution flow.

Delineating `Assert` from Conventional Assertions. The conventional `assert` statement, built into most programming languages, is a debugging aid that checks a condition and, if the condition evaluates to false, raises an `AssertionError` exception, typically terminating the program execution. In contrast, our `Assert` construct offers a sophisticated retry mechanism, while supporting a number of other new optimizations. On an `Assert` failing, the pipeline transitions to a special *retry state*, allowing it to reattempt a failing LM call while being aware of its previous attempts and the error message raised. If, after a maximum number of self-refinement attempts, the assertion still fails, the pipeline transitions to an error state and raises an `AssertionError`, terminating the pipeline. This enables `Assert` to be much more powerful than conventional assert statements, leveraging the LM to conduct retries and adjustments before concluding that an

[1] We borrow this implementation from Khattab et al. (2023). It captures the key computations in popular multi-hop question-answering systems such as Baleen (Khattab et al., 2021) and IR-CoT (Trivedi et al., 2022).

```python
class MultiHopQAWithAssertions(dspy.Module):
  ...

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

    for hop in range(2):
      query = self.generate_query(context=context, question=question).query

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

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

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

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

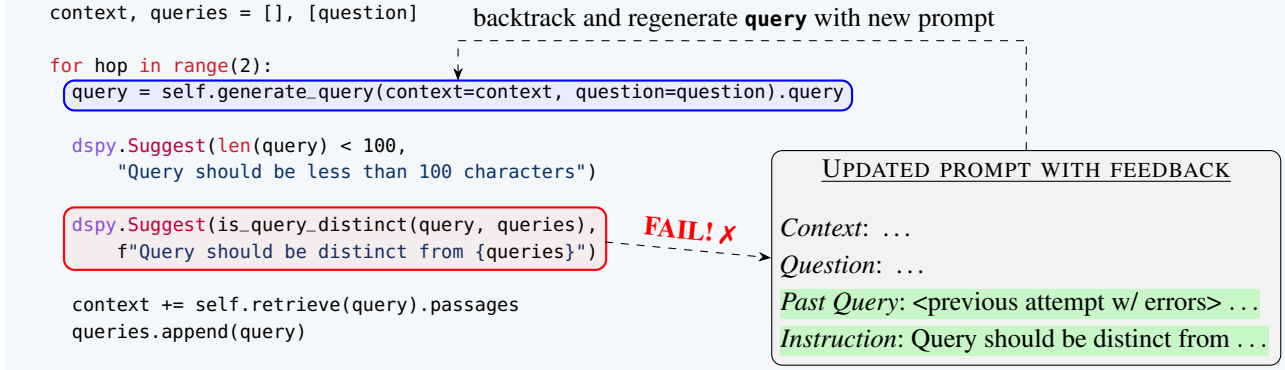

Figure 1. DSPy program with LM Assertions for multi-hop question answering task with a retriever. We introduce two suggestions: (1) the query to retriever should be shorter than 100 characters; (2) the query to retriever should differ from previous queries. Suppose the second suggestion fails, i.e., the generated query to the retriever is too similar to previous queries. Then, DSPy will construct a new prompt to retry the `generate_query` module, with additional fields indicating the previously generated query and a user-defined error message to help LM refine its generation.

error is irrecoverable.

Delineating `Suggest` from `Assert`. In contrast to our `Assert` statements, our `Suggest` statements are softer constraints that recommend but do not mandate conditions that may guide the LM pipeline toward desired domain-specific outcomes. When a `Suggest` condition is not met, similar to `Assert`, the pipeline enters the special retry state, allowing reattempts of the failing LM call and self-refinement. However, if the suggestion fails after a maximum number of self-refinement attempts, the pipeline simply logs a warning `SuggestionError` message and continues execution. This allows the pipeline to adjust its behavior in response to the suggestion while being flexible and resilient to suboptimal states (or sub-optimal or heuristic computational checks).

In the following sections, we define the default backtracking semantics of these constructs more formally. However, we provide the opportunity to extend custom semantics for both `Assert` and `Suggest` (Section 4.2).

3.1. Semantics of `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_r represents the pipeline's state, and the subscript r represents the current retry count within the state σ. The maximum number of retries allowed per assertion is denoted by R. The transitions can be defined as follows:

$$\sigma_r \vdash \texttt{Assert}(e, m) \rightarrow \sigma'_0 \quad \text{if eval}(\sigma, e) = \text{true}$$

$$\sigma_r \vdash \texttt{Assert}(e, m) \rightarrow \sigma_{r+1} \quad \text{if eval}(\sigma, e) = \text{false and } r < R$$

$$\sigma_r \vdash \texttt{Assert}(e, m) \rightarrow \sigma^{\perp} \quad \text{if eval}(\sigma, e) = \text{false and } r \geq R$$

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' and continues execution. If e evaluates to false and the current retry count r is less than the maximum allowed retries R, the pipeline transitions to a retry state σ_{r+1}. Here, the pipeline attempts to recover or adjust its behavior, incrementing the retry count r by one. If the assertion continues to fail and the retry count reaches R, the pipeline transitions to an error state $\sigma^{\perp}$, and an `AssertionError` with message m is raised, halting the execution.

3.2. Semantics of `Suggest`

The `Suggest` construct provides non-binding guidance to the LM pipeline. Similar to `Assert`, its semantics can be defined

```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 -> query")
    self.generate_cited_paragraph = dspy.ChainOfThought("context, question -> paragraph") #has field description to include citations

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

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

    pred = self.generate_cited_paragraph(context=context, question=question)
    dspy.Suggest(citations_check(pred.paragraph), "Every 1-2 sentences should have citations like 'text... [x].'")

    for line, citation in get_lines_and_citations(pred, context):
      dspy.Suggest(is_faithful(line, citation), f"Your output should be based on the context: '{citations}'.")

    return pred
```

Figure 2. DSPy program with LM Assertions 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.

as follows:

$$\sigma_r \vdash \texttt{Suggest}(e, m) \rightarrow \sigma_0' \quad \text{if eval}(\sigma, e) = \text{true}$$

$$\sigma_r \vdash \texttt{Suggest}(e, m) \rightarrow \sigma_{r+1} \quad \text{if eval}(\sigma, e) = \text{false and } r < R$$

$$\sigma_r \vdash \texttt{Suggest}(e, m) \rightarrow \sigma_0'' \quad \text{if eval}(\sigma, e) = \text{false and } r \geq R$$

If the expression e evaluates to true, the pipeline transitions to a new state σ' and continues execution. If e evaluates to false and the current retry count r does not exceed the maximum retries R, the pipeline attempts to recover or adjust its behavior in a retry state σ_{r+1}. However, different from `Assert`, if the suggestion continues to fail and the retry count r reaches R, the pipeline transitions to a new state σ'' where it resets the retry count, logs the message m as a warning of a `SuggestionError` that could not be resolved, and continues executing the next pipeline module.

3.3. Handling Self-Refinement

From the above, both `Assert` and `Suggest` allow the pipeline to retry a failing LM call and self-refine its outputs in a special retry state. One might observe that this involves dynamically altering the control flow of the LM pipeline during execution. On passing assertions and suggestions, the control flows typically into the next LM pipeline module. However, to handle failures, the LM pipeline may define an error handler that determines the next instruction to execute. The handler takes the current erring state σ and the error message m as inputs and returns a new state. In the new state, control flows as described in Section 3.1 and 3.2. For both `Assert` and `Suggest`, if the maximum retry attempts are not surpassed, the handler yields the control to the failing LM module with an updated prompt that includes past failing outputs and instructions. However, upon exceeding the maximum retries, the handler halts the execution for a failing `Assert` or progresses to the next module in the pipeline for a `Suggest`.

In the next section, we describe the implementation of these constructs and handlers in DSPy.

4. Implementation

We introduce the proposed LM Assertions 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],
            backtrack: Optional[module])
dspy.Suggest(constraint: bool, message: Optional[str],
            backtrack: Optional[module])
```

We inherit a simple API design for LM Assertions. Both suggestions and assertions take a boolean value `constraint`

as input. Note that the computation for `constraint` can invoke other DSPy modules, potentially calling the LM to inform the result for the constraint. Then, the user provides an optional error message, which is used for error logging and feedback construction for backtracking and refinement. Finally, to enable backtracking, both `dspy.Assert` and `dspy.Suggest` contains an optional `backtrack` argument, which points to the target module to backtrack to if the constraints fail.

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 encode error-handling logic as function wrappers. To that extent, we provide a primitive `constraint_tranform` to wrap any DSPy module with handlers. When the constraints in `dspy.Assert` and `dspy.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.

As a result, the program's behavior after an assertion or suggestion error is completely controlled by the handlers used. To support flexibility in using LM Assertions with DSPy, we implement several composable handlers, such as disabling suggestions and assertions, suppressing assertion errors with logging, etc.

The default handlers follow the semantics as described in Section 3 to enable self-refinement. That is, we allow R retry attempts for `AssertionError` and `SuggestionError` by backtracking to the failing LM. After R retry attempts, an `AssertionError` will be raised while `SuggestionError` will only be logged silently.

4.3. Backtracking

To implement backtracking in DSPy, we introduce a new auxiliary *meta*-module called `Retry`. This module is a lightweight wrapper for anyDSPy module, providing additional information about all previously unsuccessful predictions. When DSPy determines the need to backtrack to a specific module, it calls `Retry`. As shown in Figure 1, the `Retry` module automatically adds the failed predictions and the corresponding user-defined error messages raised to the prompt. Then, the LM pipeline can backtrack to the previously failed module with this updated prompt. In this way, the original module to refine is self-aware and informed of past attempts and errors on them. Consequently, this empowers the LM to develop more informed and error-avoiding generations in subsequent iterations of self-refinement.

5. Evaluation

In formulating and evaluating LM Assertions, we consider the following hypotheses:

H1 LM Assertions facilitate automating self-correction and refinement for arbitrary LM pipelines by showing past outputs and error messages to the LM.

H2 Self-refinement guided by LM Assertions can also enable LM pipelines to improve downstream application performance.

H3 When used with DSPy teleprompters to compile few-shot examples, LM Assertions help bootstrap more complex examples to deliver better performance for the compiled LM pipelines.

H4 Enabling LM Assertions for compilation can collect and optimize demonstrations explicitly for error-correcting behavior (for the Retry modules), which will further improve the ability of the LM application to self-refine.

In this version of our report, we focus on **H1** and **H2**. We leave the investigation for the remaining hypothesis for future versions of our paper. In this report, we explore the 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 LM Assertions at self-refining search queries for better retrieval recall as well as overall answer correctness. Next, we present results on an expanded version of the standard multi-hop QA task in a long-form QA setting. Here, the program must produce a long-form answer with citations to the retrieved context information. We evaluate on this task as an application of LM Assertions to help 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 LM Assertions specified within the program. This metric is a benchmark for the system's ability to pass internal validation checks.
- **Extrinsic Quality** assesses the impact of LM Assertions on the pipeline's performance in downstream, task-specific contexts, often ones we cannot assert directly without access to the ground-truth labels. Here, constraints act as guiding principles that indirectly influence the system's effectiveness in achieving its

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 two metrics will respectively enable us to investigate the hypotheses that LM Assertions 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 (Yang et al., 2018) dataset in the open-domain "fullwiki" setting to evaluate our downstream tasks. Since the HotPotQA test set is not publicly accessible, we reserve the official validation set for testing. 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. For training and development sets, we sample 300 examples each. Herein, we report development set scores to avoid repeated evaluation on the test set. We will include expanded test results in a future version of this report.

For retrieval, we use the official Wikipedia 2017 "abstracts" dump of HotPotQA using a ColBERTv2 (Santhanam et al., 2021) retriever. We test the program using OpenAI's `gpt-3.5-turbo` (Brown et al., 2020) language model with `max_tokens=500` and `temperature=0.7` for our experimental setup. For each task, we run two sets of experiments to test the effectiveness of LM Assertions (particularly in the form of `Suggest`), including both uncompiled and compiled modules. Uncompiled modules (`ZeroShot`) are zero-shot DSPy modules that infer the response from the questions and context directly. Compiled modules (`FewShot`) contain additional few-shot demonstrations in the prompt crafted by the DSPy compiler (Khattab et al., 2023).

5.3. Case Study: Multi-Hop QA

5.3.1. Task & Metrics

In this task, we explore complex question answering through the multi-hop program from Khattab et al. (2023). 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 intrinsic performance using a Suggestions Passed metric, which quantifies the proportion of suggestions passed across at inference time. For programs with LM Assertions, we measure the last run of the backtracking process (if any). For extrinsic quality, we use (1) Retrieval Score, which quantifies the proportion of correctly retrieved `gold` passages, and (2) Answer Correctness, which checks if the generated answer precisely matches the `gold` standard answer. Both of these are not possible to use directly at inference time, since they require access to ground-truth labels.

5.3.2. Constraints Specified

We define two simple `Suggest` statements. First, we suggest that the query generated be short and less than 100 characters, with 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 LM Assertions 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 35.7% more suggestions passed, meaning the retrieval query generation LM is more likely to generate concise, precise, and distinct queries. Then, we observe $\approx$13.3% in the retrieval score and an indirect $\approx$1.3% increase in answer correctness due to refined query generation.

We take advantage of optimizations in the DSPy compiling framework, 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

Table 1. Evaluation of MultiHop Question Answering with HotPotQA. This table presents comparisons across various 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 Suggestions Passed, Retrieval Score and Answer Correctness (Section 5.3.1)

Configuration	Suggestions Passed	Retrieval Score	Answer Correctness
`ZeroShot–NoSuggest`	64.3%	34.7%	45.7%
`ZeroShot–Suggest`	87.3%	39.3%	46.3%
`FewShot–NoSuggest`	65.0%	40.3%	49.3%
`FewShot–Suggest`	82.7%	42.0%	50.0%

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 both ≈27.2% more passed constraints and ≈1.4–4.2% on both extrinsic 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 asophisticated metric, Citation Faithfulness. In this metric, a small DSPy program uses the LM to check if the text preceding each citation appropriately supports the cited context. Our check outputs a boolean for faithfulness, which is then averaged across the citations in the output to aggregate a metric for evaluation. 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. As a simple initial check, we include a `Suggest` statement that requires every 1–2 of sentences generated has citations in an intended format. This is checked by a simple Python function `citations_check`. As a more sophisticated check, we `Suggest` 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 a small DSPy program (one that uses the 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. The goal of this `Suggest` statement is to ensure that all sentences are appropriately attributed to correct supporting sources.

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 ≈16.7% 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 ≈11.2% in citation faithfulness. However, we do see a decline in recall and precision, reflecting some com-

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, Citation Recall, Citation Precision, and Answer Correctness (Section 5.4.1)

Configuration	Citation Faithfulness	Citation Recall	Citation Precision	Answer Inclusion
`ZeroShot–NoSuggest`	76.0%	51.8%	59.4%	66.7%
`ZeroShot–Suggest`	88.7%	56.3%	64.8%	67.3%
`FewShot–NoSuggest`	86.3%	62.8%	75.3%	69.3%
`FewShot–Suggest`	96.0%	62.5%	73.8%	69.3%

plex interplay between adherence to LM Assertions 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. Programming with Constraints

Programming with constraints is standard 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.

However, most runtime checks can only be used to warn the programmer or abort the execution early. Kang et al. (2020) proposed a concept called model assertion, which can be used to monitor the behavior of machine learning models and to improve the quality of a model in training through data collection. However, among the key differences in our work is that LM Assertions abstract automatic self-refinement on violating the assertions. Our `Assert` and `Suggest` constructs can backtrack an LM pipeline to retry a failing module. On retry, additional information and feedback are automatically injected into the prompt, guiding the LM program to make better predictions to pass assertions eventually. As described in Sec 5, LM Assertions can additionally be used to inform improved compilation of DSPy programs, though we leave fully specifying and testing this to future versions of our report.

6.2. Self-Refinement and Correction

By integrating Python-style assertions, we ensure programmers can clearly express computational constraints on DSPy programs and assert desired program behavior. These declarative constraints can then be leveraged by the DSPy compiler and runtime in extensible and powerful ways to abstract and generalize notions of self-refinement and DSPy's capabilities for prompt optimization. We report on initial evaluation of an implementation that does so in this work. Such self-refinement of LLMs (Madaan et al., 2023; 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.

6.3. 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 our work 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 proposed LM Assertions (`Assert` and `Suggest`). Similarly, a synthesizer parallels an LM pipeline framework, like the DSPy compiler, striving to compile an optimized prompt.

6.4. Factual Consistency and Citations

This work evaluates LM Assertions on 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 have introduced LM Assertions, a new construct for expressing arbitrary computational constraints on the behavior of LMs when used as building blocks of larger programs. We integrate LM Assertions into the DSPy (Khattab et al., 2023) programming model, define runtime *retry* semantics, and an implementation for them that abstracts and generalizes LM self-refinement approaches to arbitrary steps in arbitrary pipelines. We also discuss several other mechanisms that our LM Assertion constructs can use to inform DSPy compilation into higher-quality prompts that reduce the assertion failure rates. Our evaluations show substantial gains on two case studies, reporting both intrinsic (i.e., assertion-specific) and extrinsic (i.e., downstream) task metrics. By enabling DSPy programs to autonomously backtrack and self-correct, we hope we have opened avenues for building more reliable LM programs at higher levels of abstraction than was previously possible.

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.

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.

Kang, D., Raghavan, D., Bailis, P., and Zaharia, M. Model assertions for monitoring and improving ml models. *Proceedings of Machine Learning and Systems*, 2:481–496, 2020.

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., Santhanam, K., Li, X. L., Hall, D., Liang, P., Potts, C., and Zaharia, M. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp, 2022.

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.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 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.

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.

Rebedea, T., Dinu, R., Sreedhar, M., Parisien, C., and Cohen, J. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails, 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.