Watch Your Steps:
Observable and Modular Chains of Thought

Anonymous ACL submission

Abstract

We propose a variant of chain of thought (CoT) prompting called Program Trace Prompting that makes explanations more observable while preserving the power, generality and flexibility of CoT. In our approach, few-shot CoT demonstrations are wrapped in a formal syntax based on Python, and CoT explanations of in-context examples are replaced with chains of these formalized steps. Each prompt also names valid steps, and describes their input/output types. By instrumenting explanations in this way, we enable new types of analysis. In particular, we identify "non-local errors" as an unaddressed issue in CoT learning, and we present methods for verifying the "modularity" of steps.

1 Introduction

While chain of thought (CoT) prompting is powerful, standard CoT outputs can be "unfaithful" (Jacovi and Goldberg, 2020): i.e., CoT can lead to incorrect but superficially plausible explanations for biased outputs (Turpin et al., 2024), and CoT explanations "may not align with . . . causal reasoning" (Bao et al., 2024). Although unfaithful explanations do not affect CoT's utility as a means of improving performance of prompted models, they do reduce the potential of CoT for other purposes, e.g., justifying a response to an end user. Despite numerous proposals (Lanham et al., 2023; Bentham et al., 2024; Parcalabescu and Frank, 2024), unfaithfulness remains difficult to detect and measure. The unfaithfulness of CoT explanations presents a jarring contrast with symbolic proofs, in which modular, verifiable reasoning steps are combined in well-understood ways.

We believe that a significant obstacle to progress on the faithfulness of CoT prompts is their syntactic diversity; i.e., because a CoT explanation can take nearly any form, they are difficult to analyze in any general way. The goal of this paper is *to make CoT explanations easier to analyze while preserving the power, generality and flexibility of CoT*. To do this, we propose a new variant of CoT prompting, in which few-shot CoT demonstrations are wrapped in a semi-formal syntax which (1) identifies and names steps; (2) defines the input/output behavior of steps; and (3) replaces each CoT explanation in a demonstration with a chain of formalized steps.

We use Python syntax to describe steps, and call our method Program Trace Prompting (PTP). As shown in Figure 1, CoT demonstrations are replaced with documentation for a Python program, together with traces of that program's behavior on the demonstration inputs. Each different kind of "step" is associated with a Python subroutine, and in addition to traces, the prompt includes *stubs* (function-level comments and type signatures) for these subroutines. An LLM is then prompted with this information as context, and asked to produce a trace for a novel input. An answer to the question is finally extracted from the LLM-generated trace.

Note that in Program Trace Prompting, *no code is presented to the LLM*. Similarly, there is no "tool use"; no generation of code or pseudo-code to be executed; and no pipeline of LLM calls, as in agent frameworks like LangChain (Topsakal and Akinci, 2023) or DsPy (Khattab et al., 2023). Instead, for a new test input, the LLM simply generates a new trace, similar to the demonstration traces, but appropriate for the test input.

Importantly, the program suggested by the traces typically do not exist, and in the more interesting applications of Program Trace Prompting, *cannot* exist as pure Python. In Figure 1, for example, the subroutine `sport_for` requires using background knowledge about arbitrary entities and phrases. While this cannot be implemented directly in Python, a strong LLM can readily perform the underlying task and thus generate a trace. Hence, the traces generated by PTP are best thought of as traces of Python-like "pseudocode" (Li et al., 2023; Weir et al., 2024).

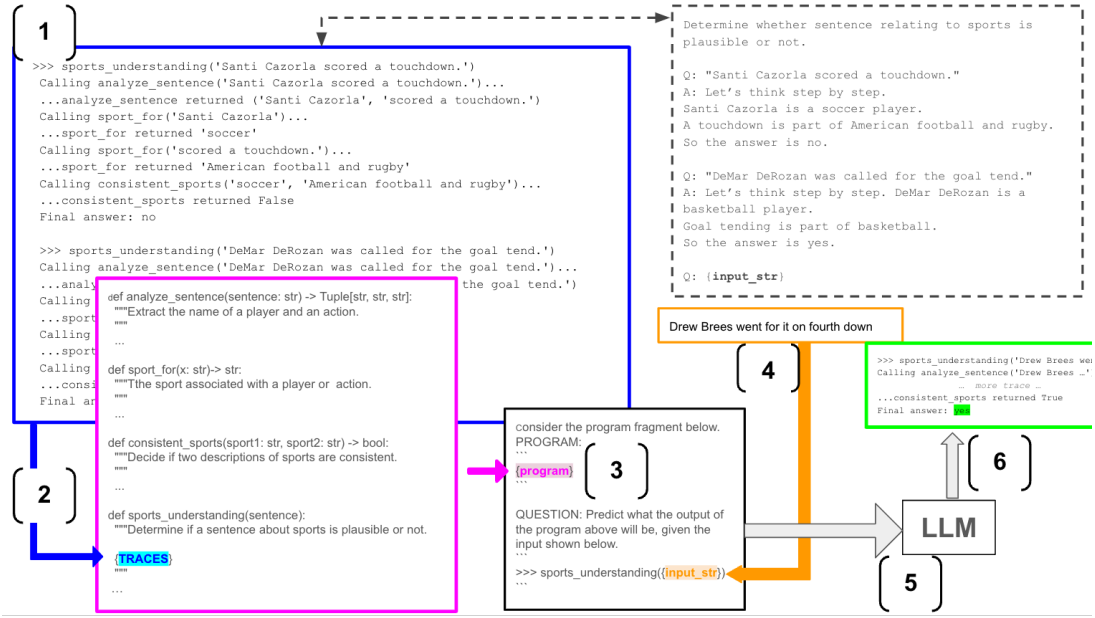

Figure 1: Program Trace Prompting on a simplified version of a BBH task. (1) The original CoT prompt is replaced with a trace of a Python pseudo-program that implements the same problem-solving strategy. (2) This trace is then combined with "stubs" describing the subroutines used in the trace. (3) This is combined with a test input (4) into a prompt that instructs the LLM to predict the output of the program on the test input. (5) This prompt is then sent to the LLM, which produces (6) a predicted program trace, here, the word "yes".

To summarize, the contributions of this paper are (1) description of a new framework for CoT prompting that allows detailed analysis of the "steps" used in a CoT explanation; (2) demonstration of the generality of this framework by testing it on 23 diverse tasks, and releasing the corresponding prompts to the community; (3) presentation of methods to evaluate two newly-defined aspects of CoT reasoning, namely the modularity of CoT steps and the locality of CoT errors; and (4) an analysis of the modularity of steps over a large set of tasks and step types.

2 Methods

2.1 Data and LLMs

To evaluate our approach, we used BIG-Bench hard (Suzgun et al., 2022), a well-studied and diverse set of 23 challenging tasks. All the tasks have 3-shot CoT prompts, and all the tasks are also well-suited to evaluation, having answers that can be easily tested for correctness (most of the tasks are multiple-choice.) About half the tasks are "NLP" tasks, and are broadly similar to the simplified example of Figure 1: a high-level reasoning strategy is followed, which requires calling low-level routines that require AI. The other half are "algorithmic" tasks, like evaluating Boolean expressions or reasoning with logical constraints.

Program Trace Prompting can be used, in principal, with any LLM. After preliminary studies, we used Anthropic's Sonnet 3 model (2024-06-20) for the experiments below (unless stated otherwise).

2.2 Constructing prompts

We constructed program trace prompts that closely followed the existing CoT prompts (as suggested by Figure 1.) The existing CoT prompts were also used as a performance baseline. Program trace prompts were produced semi-automatically. For each task, we manually wrote a Python program that implements an algorithm suggested by the CoT prompt; this is feasible because the program need work only for the few examples appearing in the CoT prompt. Following the parlance of unit testing, we call these programs *mocks*. When a mock is executed it automatically prints a trace. Traces mainly record function calls, but can also include printed output—e.g., as in Figure 1, traces contain a printed "final answer". The placement of traces (in the documentation of the top-level function) follows a Python convention for documenting function behavior by giving input/output examples.

For many tasks, the mock is a thin wrapper around the steps in the CoT demonstrations; however, some meaningful decisions do need to be made by mock authors. To avoid overfitting when

making these decisions, we used 60 random examples of each task to help design the mocks, and reserved the rest (190 examples for most tasks) as an unseen test set set. The 60 examples are further divided into "dev" and "tuning" sets with 30 examples in each. Mock implementation decisions were made based on dev data performance, but the actual values in the mocks are based on the CoT prompts, not the dev set. The tuning set was held out in early mock design for the annotation experiments of Section 3.3. Although BBH has been used for evaluating new prompting methods, there appears to be no standard dev/test split that is used to prevent developers of prompting methods from overfitting to the test data. Our splits and experimental scripts are available on GitHub[1].

2.3 Special prompting strategies

Single step prompts. One interesting aspect of Program Trace Prompting is that the same prompt can be used to either predict traces for either a full CoT inference process, or *any individual step*, by just replacing the code fragment at the end of the prompt used in stage (3) of the Program Trace Prompting process of Figure 1. For example, to predict a trace for the step `sport_for('scored a touchdown')`, one would replace the text

```
>>> sports_understanding({input_str})
```

in the stage (3) prompt with

```
>>> sport_for('scored a touchdown')
```

To our knowledge, there is no counterpart to this kind of single-step execution in conventional CoT prompting, or any of its many variants—it is a consequence of the fact that we have made steps more explicit.

Improved single-step prompts. One common failure mode for single-step prompts is that the LLM returns a trace that begins with the requested step, but then continues, usually with steps that might plausibly follow the requested step. We call this *overgeneration.* Overgeneration can be reduced by two methods. One is to add single-step traces to the stubs for the subroutine being probed. Empirically, we find that performance for single-step prompting is improved even if the only traces added are ones that are part of the top-level traces.[2] A second method is to use more a robust method for extracting output from single-step generations, i.e., parsing the trace to identify steps, and then using, as the single-step result, the return value from the first step with correct subroutine name.

Completing a partial trace. A similar prompting trick can be used to complete a trace given a prefix of the trace: we just replace the question in the stage (3) prompt with one that requests the LM to complete the trace, rather than generate it.

[1] https://github.com/REDACTED

[2] Specifically, improved performance is obtained by just copying over the appropriate part of the top-level CoT trace into the "stub" defining a step. In our experiments, we copy over the first three single-step traces for steps of interest.

	PTP		CoT
Causal Judgement	**65.8**		64.9
Date Understanding∗	88.8		**92.6**
Disambiguation QA	**82.6**		80.0
Formal Fallacies	56.8		**57.4**
Hyperbaton	**97.9**	$\gg$	85.8
Movie Recommendation	**91.1**		87.9
Penguins in a Table	89.2		**92.8**
Reasoning Colored Objs	94.2		**94.7**
Ruin Names	83.2		**86.6**
Salient Translation Errs	69.5		**70.5**
Snarks	**91.5**		89.0
Sports Understanding	**97.4**		**97.4**
Average NLP	**84.0**		83.3
Boolean Expressions	**95.3**		92.6
Dyck Languages	**76.2**	$>$	62.7
Geometric Shapes	40.0	$\ll$	**53.3**
Logical Deduction	87.9	$\ll$	**96.3**
Multistep Arithmetic	**87.9**	$\gg$	73.7
Navigate∗	**98.9**		97.9
Object Counting	**100.0**		98.9
Temporal Sequences∗	96.8		**100.0**
Tracking Shuffled Objs	98.9		**100.0**
Web of Lies	**100.0**		**100.0**
Word Sorting	**96.3**		91.4
Average Algorithmic	**88.9**		87.9
Average	**86.4**		85.5

Table 1: Accuracy on BBH problems, using a test set unseen during development of PTP prompts. Statistically significant differences are shown in the third column, with $p < 5\%$ indicated by $<$ or $>$, and $p < 1\%$ by $\ll$ or $\gg$.

3 Results: Accuracy and Types of Errors

3.1 How well do PT prompts work?

Our goal is to instrument CoT prompts so that they can be more easily analyzed—but we would like to do that without impacting the performance. Thus, Table 1 summarizes the accuracy results on the

Model	PTP	CoT
Commercial models		
Anthropic Sonnet-3.5	**92.8**	92.2
Anthropic Sonnet-3	**85.0**	83.9
OpenAI GPT4o	87.2	**90.6**
Gemini Pro-1.5	**90.6**	86.1
Open-Source models		
Deepseek Coder-v2	89.2	**92.0**
Llama-3.1 7B	64.5	**77.8**
Gemma2 9B	69.3	**82.2**

Table 2: Average results with other LLMs on a subset of tasks.

	Avg
Call correct	99.3%
Syntactically correct	98.6%
Type correct	98.6%

Table 3: Overall correctness of calls, argument/return value syntax, and argument/return value types on the tune set for 23 BBH tasks.

BBH tasks.[3] The results of Table 1 strongly support our first claim, that *PTP is applicable to a broad range of tasks*, in the sense that it can be used on a broad range of tasks without a major impact in performance. Of the 23 diverse tasks in BBH, only five had statistically significantly different performances, and there is no clear pattern among these. The average performance of PTP is slightly better than CoT.

In Table 2, we also show results comparing PTP and CoT prompting for several other strong LLMs, including one strong open-weight model (Deepseek Coder V2, a 236B model). To reduce costs these experiments were run on the dev set for six tasks[4] chosen for diversity. For most strong models the average performance is similar (within 3-4 points) for both prompting methods. However, PTP clearly underperforms CoT prompting on smaller models.

3.2 Are generated traces well-formed?

The main goal of Program Trace Prompting is to produce structured explanations, and thus enable deeper analysis of CoT performance. To verify structure, we wrote tools to evaluate generated traces for the following properties. *(1) Call correctness:* traces were parsed to see if they are well-formed trace[5] and only contain calls to given functions. *(2) Syntactic correctness:* for well-formed traces, the arguments and return values were syntactically checked with a Python parser. *(3) Type correctness:* syntactically correct arguments and return values were type-checked using the type hints given in the stubs.

The analyses of this paper only require call correctness, but interestingly, Program Trace Prompting generally produces traces that are syntax- and type-correct as well. Table 3 gives a summary average over the tune set on tasks. This analysis supports our second claim, that *the traces generated by PT prompts can almost always be automatically parsed into legal sequences of the predefined steps associated with the prompt.* This allows us to further analyze the traces and characterize their errors.

```
Calling sport_for('Santi Cazorla')...
...sport_for returned 'soccer'
Calling sport_for('scored a touchdown.')...
...sport_for returned 'US football / rugby'
Calling consistent_sports('soccer', 'US football / rugby')...
...consistent_sports returned False
```

```
Calling sport_for('Santi Cazorla')...
...sport_for returned 'rugby'
Calling sport_for('scored a touchdown.')...
...sport_for returned 'US football / rugby'
Calling consistent_sports('rugby', 'US football / rugby')...
...consistent_sports returned True
```

```
Calling sport_for('Santi Cazorla')...
...sport_for returned 'soccer'
Calling sport_for('scored a touchdown.')...
...sport_for returned 'US football / rugby'
Calling consistent_sports('rugby', 'US football / rugby')...
...consistent_sports returned True
```

Figure 2: Top, part of a correct program trace. Middle, a *local* error: the first `sport_for` step returns an incorrect result (red), which causes an incorrect answer. Bottom, a *non-local* error: the `consistent_sports` call should have the bold-faced arguments 1 and 2 copied over from the first and second `sport_for` outputs, respectively, but the red bold-faced argument was not copied correctly.

3.3 Local and non-local errors

Since the PTP prompt specifies the semantics of individual steps, is possible to annotate *where* errors occur. In general, errors in PTP reasoning can occur either because a step produces a wrong output (which we call here a *local error*), or because a wrong sequence of steps is used (which we call a *non-local error*). Examples of these error types

[3] To avoid prompt-tuning that might overfitthe results in the table are generally for the *first* PT prompt for a task that was evaluated on the test data. For the three starred cases, problems discovered were in the PTP prompt late in experimental procedure, and the results are for the second prompt tested.

[4] Date Understanding, Hyperbaton, Sports Understanding, Geometric Shapes, Multistep Arithmetic Two, Logical Deduction Three Objects.

[5] E.g, every `"Calling f(a, b)..."` line can be paired with a corresponding `"...f returns c"` line.

	% of Traces	% of Errors
All errors	11.7	100.0
Local errors	9.1	77.8
Non-local errors	2.6	22.2

Table 4: Analysis of local-vs non-local errors in BBH hard tasks on the tuning set.

Task	Avg # Steps	Trace Ent.	Non-Local Errs
object counting	3.0	0.00	
nine low-ent. tasks	...	...	
sports understanding	4.3	0.44	
multistep arithmetic	26.2	1.09	3.3%
ruin names	18.4	1.14	
date understanding	11.3	1.19	3.3%
penguins in a table	3.6	1.20	3.3%
hyperbaton	11.5	1.59	
causal judgement	6.0	1.76	
geometric shapes	17.1	1.76	13.3%
navigate	7.3	1.98	
dyck languages	42.7	4.00	23.3%
formal fallacies	12.3	4.15	13.3%
boolean expressions	7.4	4.33	
word sorting	47.8	4.83	

Table 5: Number of steps and trace abstraction entropy of the 23 BBH Tasks (on the test data). Lower trace entropy indicates a more predictable sequence of steps. Only non-zero non-local error rates are shown.

are illustrated in Figure 2; as suggested, the most common non-local error involves copying values from the wrong place.

We manually annotated all of the traces from the tune set that led to an incorrect final result. We marked the first[6] incorrect step, if there was one, and otherwise marked the trace as containing a non-local error. The results are shown in Table 4.

To our knowledge, the existence of non-local errors in CoT reasoning has not been discussed previously in the literature. Despite this, non-local errors are not unimportant: 22% of the incorrect traces were annotated as having non-local errors.

Our conjecture is that non-local errors are correlated with the difficulty of learning to sequence the steps: in other words, *non-local errors arise when the algorithm suggested by the CoT trace is complex, and hence difficult to learn.* To test this, we

[6]Only the first local error was marked, because errors often cascade in CoT reasoning chains.

Task Step	Single-step Accuracy
tracking shuffled objs	
simulate_swap	96.7
answer_question	100.0
word sorting	
kth_letter	96.7
partition_words	100.0
sort_keys	100.0
bucket_sort	76.7
flatten	100.0
multistep arithmetic	
is_simple_expr	83.3
rewrite_expr	73.3
eval_simple_expr	100.0
eval_expr	90.0
eval_variabilized_expr	100.0
dyck languages	
is_open_paren	100.0
update_stack	80.0
empty_stack	86.7
boolean expressions	
solve_negation	73.3
Average	91.0
Average with ablations:	
no 1-step traces	87.9
no trace parsing	86.9

Table 6: Testing the accuracy of single-step prompting, on 16 steps from 5 tasks that can be checked against an oracle.

analyzed the traces produced for each task on the test examples. For every legal trace, we simplify the trace by removing the arguments and return values, leading to an *abstract trace*. To quantify the algorithm complexity, we computed the entropy of the empirical distribution of abstract traces for each task, which we show in Table 5. *The correlation between non-local error rate and trace entropy is* $r = 0.51$. In contrast, the correlation to the number of steps in the trace (also shown) is only $r = 0.04$.

4 Results: Single-step prompts

4.1 Evaluation with oracles

We now turn to the question of how reliably LLMs can perform individual steps, using the prompting scheme of Section A.3. Evaluating performance on a single step is difficult, since we do not have gold labels for each step's output. Fortunately, several individual steps in the BBH tasks are possible to implement in Python, and the local correctness of these steps can be readily evaluated by using the corresponding Python function as an oracle.

We implemented Python oracles for 16 different steps from five tasks, and evaluated these oracle-testable steps on the inputs that were actually used in the course of solving the dev set examples. Ta-

Prompt	Interv.	Acc	Agree	#Steps
baseline	no	97.4		
changed	no	98.4		4.30
changed	yes	97.4	56.8	4.65

Table 7: Experiments with version of `sports_understanding` with a step that has been modified to be non-modular.

ble 6 shows that the overall performance is more than 90%. Here we use the full method described in Section A.3, including the two methods used to address overgeneration. Table 6 also shows the result of ablating these extensions, showing they contribute to performance.

4.1.1 Evaluation without oracles

A flawed approach. All of the oracle-testable steps are used in algorithmic tasks: in general devising oracles is difficult because step output can be natural-language text whose utility for downstream processing is uncertain. We now turn to a more difficult question: how can we measure performance of single-step prompting when no oracle exists?

A natural approach would be to measure the effectiveness of single-step output when used as part of a problem-solving episode—i.e., to consider a single-step output as "correct" if it leads to a correct final answer, rather than if it agrees with the oracle. (Approaches like this have been used for verifying steps in math tasks (Wang et al., 2023a; Viteri et al., 2024)).

Specifically, we (1) identify the places in trace t where some particular step i of interest is used; (2) get the single-step prompted output $\tilde{y}_i$ for step i and replace the old step-i output y_i with $\tilde{y}_i$; and finally (3) complete the trace prefix up through $\tilde{y}_i$ using the method of Section 2.3, yielding a new trace $\tilde{t}$. We perform steps (1-3) for each trace $t \in T$, where T is a set of traces, yielding a set set $\tilde{T}$, and compare the accuracies of $\tilde{T}$ and T. Following the terminology of (Lanham et al., 2023), we call this process an *intervention*.

A simple experiment shows this approach can yield very unintuitive results. We synthetically produced a step that would be impossible to perform accurately in isolation, by removing the first input argument of the `consistent_sports` step in the `sports_understanding` task.[7] One might expect this prompt to be less accurate, but evaluated it in the usual way, accuracy is statistically identical to the baseline PTP prompt, as shown in the first two lines of Table 7. The reason, of course, is that the LLM is not restricted to generate step outputs from the strings marked as step inputs—in generating step outputs, the LLM can attend to *any* previous text.

We then intervene following the 3-step process above, thus *forcing* the LLM to consider only the designated inputs to this step. Surprisingly, this still makes no significant difference in accuracy (last line of Table 7)!

Inspection shows why this happens. Performed in isolation, the modified `consistent_sports` step agrees with the in-context step only 56.8% of the time—but in stage 3, when regenerating the remainder of the trace, the LLM often adds an extra `consistent_sports` call, which (performed in context) corrects the introduced error. (Similar behavior has been observed elsewhere (Yee et al., 2024a,b).) The intervention thus leads to similar accuracy—but performs more steps on average.

An effective approach. This suggests evaluating single-step prompts with statistics other than accuracy (e.g., trace length), so as to detect LLM corrections.

Notice that this new experiment doesn't measure the accuracy of single-step prompting—rather it measures if a step behaves *differently* with a single-step prompt than it does in a complete trace. We call this new property *modularity*, and say that a step is *non-modular* if the single-step prompt behaves differently, and is *modular* otherwise.

To measure modularity of step i on a trace sample T, we perform the intervention above to produce $\tilde{T}$. Let $g(t)$ be some summary statistic derived from a trace, e.g., the length or correctness of the trace, and let S, $\tilde{S}$ be the result of computing g on T, $\tilde{T}$ respectively. To detect modularity, we look for changes in the distribution of $\tilde{S}$ relative to S. In addition to trace correctness g_{corr}, we consider two additional statistics: the number of steps in the trace, g_{steps}, and the abstract trace of Section 3.3, g_{trace}. We detect changes between $\mathcal{S}$ and $\tilde{\mathcal{S}}$ with a sign test for g_{corr} and a paired t-test for g_{steps}. For g_{trace}, we test if the Jensen-Shannon divergence between S and $\tilde{S}$ is higher than expected using a permutation test.

One subtlety is that the intervention consists of two changes: replacing step i (stage 2 of the intervention), and regenerating the trace after step i (stage 3), and as defined modularity refers the effects of stage 2 only. To tease these two issues

[7] Recall that this step compares two sports descriptions, which clearly cannot be done without seeing the first one.

Task step	P(reject null) g_{steps}	g_{trace}	g_{corr}
causal judgement			
plausible_conclusion	1.00	0.46	1.00
plausible_inference	1.00	0.35	1.00
relevant_sentences	1.00	1.00	1.00
date understanding			
answer_question	1.00	0.41	1.00
make_inference	1.00	0.27	1.00
disambiguation qa			
find_possible_interps	1.00	1.00	1.00
is_interp_logical	1.00	1.00	1.00
formal fallacies			
to_logical_form	0.76	0.95	1.00
prove: **mod unclear**	1.00	**0.05**	1.00
(testing S vs. $\bar{S}$)	***0.08***	*0.41*	*0.63*
hyperbaton			
classify_adjective	0.66	0.24	1.00
movie recommendation			
explain_best_choice	1.00	1.00	1.00
movie_properties	0.66	0.97	1.00
summarize_movies	1.00	1.00	1.00
penguins in a table			
answer_question	0.98	0.99	1.00
table_operation	0.98	1.00	1.00
reasoning @ colored objs			
query_colored_objects	1.00	1.00	1.00
ruin names			
humorous_edit	0.98	0.950	1.00
meaningful_edit	0.98	0.963	1.00
salient translation			
choose_answer	1.00	1.00	1.00
choose_err_type: **not mod**	**0.00**	**0.05**	1.00
(testing S vs. $\bar{S}$)	*0.33*	*1.00*	*1.00*
find_trans_err: **not mod**	**0.00**	**0.05**	1.00
(testing S vs. $\bar{S}$)	*1.00*	*1.00*	*1.00*
snarks			
is_sarcastic	1.00	1.00	1.00
judge_statement	1.00	1.00	1.00
summarize_statement	1.00	1.00	0.56
sports understanding			
consistent_sports	1.00	1.00	1.00
sport_for	1.00	1.00	1.00

Table 8: Testing non-modularity and incremental generation assumption on selected NLP tasks. (See text)

apart, we consider a simpler intervention where one simply truncates the trace after step i, and then regenerates the rest. Below we use $\bar{T}$ and $\bar{S}$ for the results of this intervention.

In Table 8, we evaluate nodularity for 25 steps from 12 NLP tasks (we choose steps which were prone to local errors as being relatively more interesting), recording the p-value of the null hypothesis[8] that S and $\tilde{S}$ are the same, and boldfacing values where $p < 0.05$ (after rounding). Only 3 of these steps show a significant distributional shift after intervention. For these three we additionally compare S and $\bar{S}$, showing that the distributional shift for one of the three steps (`prove`) may be explained by difficulties performing stage 3 of the intervention.[9]

We conclude that non-modularity is rare in NLP tasks.

5 Related Work

Measuring faithfulness of CoT. The notions of modularity and locality explored here are related to explanation *faithfulness* (Jacovi and Goldberg, 2020; Turpin et al., 2024; Lanham et al., 2023). Turpin *et al* noted that when LLMs are biased (e.g., illustrating a gender bias in predictions of occupation), CoT prompting sometimes leads to predictions that preserve the underlying bias, and produces plausible-looking CoT explanations that do *not* reflect the underlying bias. This sort of "unfaithful" explanation is disturbing, perhaps because it seems like deception. Later work (e.g., (Bao et al., 2024)) support this. Some proposals to measure faithfulness even adopt very general approaches for numerically measuring influence in neural networks (Parcalabescu and Frank, 2024) to address perceived limitations in measuring faithfulness (Bentham et al., 2024).

Here we build on the work of (Lanham et al., 2023), where faithfulness is measured by changing a generated explanation (e.g., introducing errors, paraphrasing, etc) and then measuring the impact on accuracy. We extend this work by considering a particular change (forced modularity of a step); using new statistics to measure the impact of a change; and studying the more precise notions of modularity of steps and error locality. Our work is also influenced by (Yee et al., 2024b,a), who explore error recovery in CoT models and its effect on measuring faithfulness.

Prior work has developed techniques for verifying the correctness of steps in math (Viteri et al., 2024; Wang et al., 2023a) and theorem-proving tasks (Saparov and He, 2022; Lai et al., 2024) based on domain-specific step classifiers, or domain-specific classifiers of inference chains (Paul et al., 2024; Sun et al., 2024). Our modularity test differs in that it is general enough to apply to many NLP CoT tasks, and in that it tests for the related property of modularity.

Calling Python from LLMs. One method that has been proposed to improve faithfulness, and more generally to enhance reasoning abilities, is

[8] After multiple test correction.

[9] This is one of the tasks with a high trace entropy, and high rate of non-local errors.

to combine LLMs with symbolic methods–often methods based on Python. For example, in Program of Thoughts (Chen et al., 2022), Program Aided LMs (Gao et al., 2023) and elsewhere (Lyu et al., 2023), an LLM generates a Python program for each task instance, which is then executed to produce an answer. An alternative to this architecture is tool use (Paranjape et al., 2023), and a third architectural alternative is to generate Python-like "pseudocode" (Li et al., 2023; Weir et al., 2024; Chae et al., 2024)—code which has a symbolic, Python-like syntax, but includes routines that cannot be executed directly, but must be emulated by an LLM, or Python code which uses LLMs as a tool (Zhou et al., 2024). Of the approaches above, the "pseudocode" architecture is closest to Program Trace Prompting, but past work in this line has sought to produce a single program that works for all instances of a task. In contrast, our approach does not produce a program, or even require that such a program exists, since a trace is generated for each task instance. A second difference is that unlike the prior work discussed above, Program Trace Prompting does not use a hybrid LLM/symbolic architecture, but is a pure prompting method, thus providing an improvement in simplicity.

Calling LLMs from Python. Another widely-used methodology for developing applications with LLMs is to manually decompose a task into substeps that can be performed by making a series of calls to LLMs. Examples of software systems that support this methodology include DsPy (Khattab et al., 2023) and LangChain (Topsakal and Akinci, 2023). Unlike agentic programs, Program Trace Prompting performs this sort of decomposition within a single prompt.

CoT Prompt generation. Past work has sought to reduce the cost of producing (Shao et al., 2023) and optimizing (Wang et al., 2023b) CoT prompts, including proposals for semi-automatic CoT prompting by making use of post-hoc extractive rationales of classification decisions (Krishna et al., 2024). Program Trace Prompting also proposes semi-automatic production of CoT prompts, but using approaches are specific to generation of PT prompts.

Learning programs from traces. Learning from program traces has been studied for many decades in machine learning (e.g., (Smith, 1984)), and has gone by various names including programming by example and learning by demonstration (Cypher and Halbert, 1993; Lieberman, 2001; Lau et al., 2003). To our knowledge, there has been little work on this topic in the LLM era, although algorithmically produced programs traces have been used to analyze the noise-sensitivity of CoT reasoning (Havrilla and Iyer, 2024).

6 Conclusions

We propose Program Trace Prompting, a variant of CoT prompting in which few-shot CoT demonstrations are wrapped in a semi-formal syntax which (1) identifies and names steps; (2) defines the input/output behavior of steps; and (3) replaces CoT explanations of in-context examples with chains of these formalized steps on the same example. We show that this approach is broadly applicable, and generally comparable in accuracy to CoT prompting. It also offers some advantages over prior prompting schemes.

First, Program Trace Prompting traces are *easier to analyze automatically than arbitrary CoT explanations*, since they follow a defined syntax—e.g., we can extract inputs and outputs of individual steps, count the number of inference steps, and perform other types of analysis easily. More interestingly, we can localize errors in an incorrect explanation, and identify errors which cannot be localized to specific steps (e.g., copying errors where the input to one step should be output of another, but is not). We showed that *trace entropy* correlates with the presence of non-local errors: simply put, more complicated reasoning strategies are harder to learn from CoT prompts.

Second, Program Trace Prompting allows prompts to *elicit results of executing a single step of a CoT process in isolation*, outside of a larger reasoning task. This enables a new class of interventions, where we force a step inside a trace to be *modular* by executing it in isolation, as if it were performed by an agentic LLM program. By doing this and measuring statistical perturbations in the result, we can detect if the step actually depends only on its declared inputs, or if it depends on other previous steps in unexpected ways. We show that this sort of non-modular behavior can occur, but it is rare in our collection of tasks.

Although modularity has not been explored previously, we believe that this property is intuitively important for any process that claims to solve problems "step by step".

7 Limitations

When we compare Cot and PTP, we discard examples that fail to generate a clear final answer. However, Program Trace Prompting generations are longer than the baseline CoT prompts, which makes them more expensive to generate, and sometimes impossible to generate with one LLM call. They are also more likely to be blocked for safety reasons (perhaps because re-expressing an unsafe prompt as code generation is sometimes proposed as a way of bypassing safety features.)

Our approach to detecting non-modularity is based on looking for statistical differences (rejecting a null hypothesis) in summary statistics. If non-modularity is *not* detected in an experiment, it might still be detectable with additional examples, or a different summary statistic.

Our analysis of locality of errors and modularity were conducted with only one LLM. Although other strong LLMS do seem to support PTP, we do know if their behavior with respect to locality and modularity will be consistent.

We have presented only one implementation of PTP, and do not know if other variants would be more or less effective (e.g., using different base programming languages, different textual representations for traces, etc).

8 Ethical Impact

"Unfaithful" CoT explanations can be easily mistaken as either a causal explanation of an LLM's underlying generation process, or as a meaningful justification of a response. Actually providing such causal explanations or justifications is very difficult, but we have made some small steps toward this by making explanations easier to analyze. Although this does have benefits—e.g., identifying a new type of "non-local" error in CoT explanations, and identifying new tools for detecting unexpected long-range dependencies in CoT steps ("non-modularities"—it is still possible for generated traces to be misinterpreted as causal explanations or justifications for a response.

One conjecture motivating this work is that PTP will lead to a more precise way of controlling CoT prompting—or, if this is not possible, ways of detecting inputs for which LLMs use unexpected and therefore less-reliable reasoning strategies. Although PTP does seem to follow demonstrations more closely than traditional CoT prompts do, this conjecture has yet to be quantified and tested.

References

Guangsheng Bao, Hongbo Zhang, Linyi Yang, Cunxiang Wang, and Yue Zhang. 2024. Llms with chain-of-thought are non-causal reasoners. *arXiv preprint arXiv:2402.16048*.

Oliver Bentham, Nathan Stringham, and Ana Marasović. 2024. Chain-of-thought unfaithfulness as disguised accuracy. *Preprint*, arXiv:2402.14897.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu-iunn Ong, Beong-woo Kwak, Moohyeon Kim, Seonghwan Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, et al. 2024. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. *arXiv preprint arXiv:2404.02575*.

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

Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Alex Havrilla and Maia Iyer. 2024. Understanding the effect of noise in llm training data with algorithmic chains of thought. *Preprint*, arXiv:2402.04004.

Alon Jacovi and Yoav Goldberg. 2020. Towards faithfully interpretable NLP systems: How should we define and evaluate faithfulness? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4198–4205, Online. Association for Computational Linguistics.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Satyapriya Krishna, Jiaqi Ma, Dylan Slack, Asma Ghandeharioun, Sameer Singh, and Himabindu Lakkaraju. 2024. Post hoc explanations of language models can improve language models. *Advances in Neural Information Processing Systems*, 36.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv preprint arXiv:2406.18629*.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, et al. 2023. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*.

Tessa Lau, Pedro Domingos, and Daniel S Weld. 2003. Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on Knowledge capture*, pages 36–43.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*.

Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *Preprint*, arXiv:2301.13379.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.

Letitia Parcalabescu and Anette Frank. 2024. On measuring faithfulness or self-consistency of natural language explanations. *Preprint*, arXiv:2311.07466.

Debjit Paul, Robert West, Antoine Bosselut, and Boi Faltings. 2024. Making reasoning matter: Measuring and improving faithfulness of chain-of-thought reasoning. *arXiv preprint arXiv:2402.13950*.

Abulhair Saparov and He He. 2022. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 30706–30775. PMLR.

Douglas R Smith. 1984. The synthesis of lisp programs from examples: A survey. *Automatic program construction techniques*, 307:324.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. 2024. Easy-to-hard generalization: Scalable alignment beyond human supervision. *arXiv preprint arXiv:2403.09472*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Oguzhan Topsakal and Tahir Cetin Akinci. 2023. Creating large language model applications utilizing LangChain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pages 1050–1056.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. 2024. Language models don't always say what they think: unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36.

Scott Viteri, Max Lamparth, Peter Chatain, and Clark Barrett. 2024. Markovian agents for truthful language modeling. *arXiv preprint arXiv:2404.18988*.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Y Wu, and Zhifang Sui. 2023a. Math-shepherd: A label-free step-by-step verifier for llms in mathematical reasoning. *arXiv preprint arXiv:2312.08935*.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P. Xing, and Zhiting Hu. 2023b. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *Preprint*, arXiv:2310.16427.

Nathaniel Weir, Muhammad Khalifa, Linlu Qiu, Orion Weller, and Peter Clark. 2024. Learning to reason via program generation, emulation, and search. *arXiv preprint arXiv:2405.16337*.

Evelyn Yee, Alice Li, Chenyu Tang, Yeon Ho Jung, Ramamohan Paturi, and Leon Bergen. 2024a. Dissociation of faithful and unfaithful reasoning in llms. *arXiv preprint arXiv:2405.15092*.

Evelyn Yee, Alice Li, Chenyu Tang, Yeon Ho Jung, Ramamohan Paturi, and Leon Bergen. 2024b. Faithful and unfaithful error recovery in chain of thought. In *First Conference on Language Modeling*.

Ben Zhou, Hongming Zhang, Sihao Chen, Dian Yu, Hongwei Wang, Baolin Peng, Dan Roth, and Dong Yu. 2024. Conceptual and unbiased reasoning in language models. *Preprint*, arXiv:2404.00205.

A Details on Methods

A.1 CoT Prompts

Modifications to CoT prompts were made only (1) to encourage the model to closely follow the in-context demonstrations, and in particular to produce output in an easily-parsed format and (2) rarely, to fix reasoning errors in the CoT prompt.

The original CoT prompts from (Suzgun et al., 2022) are designed to give an explanation, followed by a short answer which can be checked against a target. Modern RLHF- and instruction- trained models do not always follow the syntax of CoT examples, so we modified the prompts as suggested by the example below, where the italicized text was added:

> Determine whether an artificially constructed sentence relating to sports is plausible or not.
>
> *When you give your answer, follow the format of the examples below carefully. In particular, you MUST end your answer with either 'Final answer: yes' or 'Final answer: no'.*
>
> Q: Is the following sentence plausible? "Bam Adebayo scored a reverse layup in the Western Conference Finals."
>
> A: Let's think step by step. Bam Adebayo is an American basketball player. Scoring a reverse layup in the Western Conference Finals is part of the NBA Finals. So the answer is yes.
> *Final answer: yes*

For multiple-choice questions, the new instruction says "*... you MUST end your answer with the line 'Final answer: (X)' for some letter X.*"

In addition to making it easier to recognize the answer, the instruction to *follow the format of the examples carefully* also encourages CoT reasoning and makes the CoT baseline more comparable to the PTP method.

Because robustness to noisy examples is not the focus of this paper, we also corrected two errors in the CoT examples: a set of incorrect coordinates in `navigate`, where positive and negative examples were swapped, and one movie name in `movie recommendation` which was broken across two options ("They Shoot Horses Don't They" was split into two movie choices, "(A) They Shoot Horses" and "(B) Don't They".) The prompts used are available in https://github.com/REDACTED under `modified-cot-prompts`.

A.2 Generation of traces

In our implementation, mocks are special Python programs, and invoking a mock program from the command-line with appropriate arguments will produce the "program" shown in Step (1) of Figure 1 of the main paper. Mock programs can include preprocessing directives—e.g., code that is not surrounded by the lines `###IF prompt` and `###ENDIF prompt [ELSE ...]` will not be included in the Step (1) program. Other preprocessing directors specify where to insert generated traces (e.g., `###DOCTESTS FOR foo`). The "stub" for a step can be linked to a mock implementation by decorators like `@dictmock` for a dictionary-based implementation and `@proxymock` for a Python function implementation. A similar decorator associates a top-level function with the inputs used to automatically generate the traces/demonstrations from the mock. Figure 3 shows an example of a simple mock and the corresponding generation.

A.3 Prompts for generated traces and continuing partial traces

To predict a trace for the step `sport_for('scored a touchdown')`, one could replace the text

```
>>> sports_understanding({input_str})
```

in the stage (3) prompt with

```
>>> sport_for('scored a touchdown')
```

An ideal generation for the prompt above might be something like

```
Calling sport_for('scored a touchdown')...
...sport_for returned 'American football'
'American football'
```

The final output of the step is then simply the last line of the generation.

The result of single-step prompting is more variable than the result of PTP prompting on full examples. One common failure mode for single-step prompts is that the LLM returns a trace that begins with the requested step, but then continues, usually tracing steps that might plausibly follow the requested step. We call this *overgeneration.*

Overgeneration can be reduced by two methods. One is to add single-step traces to the stubs for the subroutine being probed. Empirically, we find that performance for single-step prompting is improved even if the only traces added are ones that are part of the top-level traces. Specifically, improved performance is obtained by just copying over the appropriate part of the top-level CoT trace into the "stub" defining a step, as discussed in Section A.4. A second method is to use more a robust method for extracting output from single-step generations, i.e., parsing the trace to identify steps, and then using, as the single-step result, the return value from the first step with correct subroutine name.

Task	Description
NLP Tasks	
Causal Judgment	Answer questions about causal attribution
Date Understanding	Infer a date from context
Disambiguation QA	Clarify the meaning of sentences with ambiguous pronouns
Formal Fallacies Syllogisms Negation	Distinguish deductively valid arguments from fallacies
Hyperbaton (Adjective Ordering)	Order adjectives correctly in English sentences
Movie Recommendation	Recommend movies similar to a given list of movies
Penguins in a Table	Answer questions about a table of penguins and their attributes
Reasoning about Colored Objects	Answer questions about the colors of objects on a surface
Ruin Names	Select the humorous edit that 'ruins' a movie or musical artist name
Salient Translation Error Detection	Detect the type of error in an English translation of a German sentence
Snarks	Determine which of two sentences is sarcastic
Sports Understanding	Determine whether a sentence relating to sports is plausible or not
Algorithmic Tasks	
Boolean Expressions	Evaluate the result of a random Boolean expression
Dyck Languages	Correctly close a series of open/close parenthesis of different types
Geometric Shapes	Name geometric shapes from their SVG paths
Logical Deduction	Deduce the order of a sequence of objects given constraints
Multi-Step Arithmetic	Solve multi-step arithmetic problems
Navigate	Given navigation instructions, determine whether one ends up back at the starting point
Object Counting	Count the number of objects of different types
Temporal Sequences	Answer questions about which times certain events could have occurred
Tracking Shuffled Objects	Find the final positions of objects given initial positions and a sequence of swaps
Web of Lies	Evaluate a random boolean function expressed as a word problem
Word Sorting	Sort a list of words

Table 9: Tasks in BIG Bench Hard. For each task, there a 3-shot CoT prompt and 250 test examples (except for "Snarks", which has only 178 examples, and "Causal Judgement", which has only 187.)

The prompts to generate a trace, and to continue a partial trace, are shown in Figure 4.

A.4 Single-step prompts

The standard prompt to generate a trace was used, but we address over-generation in two ways, as discussed.

- Trace parsing: To extract the output of the step f_i, we analyze the trace to find all pairs of matching lines of the form `Calling` $f_i(x_i)\ldots$ and $\ldots$ f_i `returned` y_i. We then use the string y_i for the first of these pairs as the output.

- Micro-traces: For the steps f_i for which we plan to perform single-step prompting, we insert into the stub for f_i the preprocessing directive `###DOCTESTS FOR` f_i `IMPLIED BY` h, where h is the name of the top-level mock function (and also the name of the task.) At program generation time, this directive trace-parses the generated traces for h, the top-level function, and detects all of the `Calling`/`returned` line pairs for f_i. The first $K = 3$ of these pairs (together with all lines between a pair) are inserted in place of the preprocessing directive.

B Experimental details

B.1 Avoiding overfitting to the test data

To avoid overfitting, only one PT prompt was evaluated on the test set in Table 1 of the main paper, except for three cases in which problems discovered in the PTP prompt late in experimental procedure. For these the PT prompt result of Table 1 is the second prompt evaluated.

One case (Navigation) had a bug in which intermediate results were not printed in some cases. In two cases (Date Understanding and Temporal Sequences) review of the mock algorithm determined it was quite different from a CoT-suggested method, and the PT prompt was replaced with one following a more similar strategy. Recall the goal of this paper is exploring new ways to observe and analyze CoT explanations, not to explore performance differences on these specific tasks; hence we want to align the algorithms implied by the PT prompts with those implied by the CoT prompts as much as possible.

B.2 Comparing CoT and PTP

In the experiment of Table 1 of the main paper, it is possible for models to consistently fail on an example. The most common reasons for this are that the LLM's "guard rails" are triggered and it refuses to produce output (e.g., some of the Causal Rea-

```python
import mock

#
# implement the mocks
#

INPUT1 = "Santi Cazorla scored a touchdown."
INPUT2 = "DeMar DeRozan was called for the goal tend."

PLAYER1, ACTION1  = 'Santi Cazorla', 'scored a touchdown.'
PLAYER2, ACTION2 = 'DeMar DeRozan', 'was called for the goal tend.'

INPUTS = [INPUT1, INPUT2]

analyze_sentence_dict = {
    INPUT1: (PLAYER1, ACTION1),
    INPUT2: (PLAYER2, ACTION2),
}

sport_for_dict = {
    # answers for the two CoT examples
    PLAYER1: 'soccer',
    ACTION1: 'American football and rugby',
    PLAYER2: 'basketball',
    ACTION2: 'basketball',
}

def consistent_sports_proxy(sport1, sport2):
    # this function words on the two CoT examples
    return sport1 == sport2

#
# Prompt is postprocessed version of code betweeen 'IF prompt' and
# 'ENDIF prompt', with traces for function on each sample_input
# inserted where 'DOCTESTS for f' appears
#

###IF  prompt

from typing import Tuple

@mock.dictmock(analyze_sentence_dict)
def analyze_sentence(sentence: str) -> Tuple[str, str]:
  """Extract the player name and action from a sentence about sports.
  """
  ...

@mock.dictmock(sport_for_dict)
def sport_for(x: str)-> str:
  """The name of the sport associated with a player or action.
  """
  ...

@mock.proxymock(consistent_sports_proxy)
def consistent_sports(sport1: str, sport2: str) -> bool:
  """Test if two strings describing sets of sports overlap semantically.
  """
  ...

@mock.nullmock(sample_inputs=INPUTS)
def sports_understanding(sentence):
  """Determine if a sentence about sports is plausible or not.

  ###DOCTESTS FOR sports_understanding
  """
  ###IF toplevel ELSE ...
  player, action = analyze_sentence(sentence)
  player_sport = sport_for(player)
  action_sport = sport_for(action)
  result = consistent_sports(player_sport, action_sport)
  print(f'Final answer: {"yes" if result else "no"}')
  return result
  ###ENDIF toplevel
###ENDIF prompt
```

```python
from typing import Tuple

@traced
def analyze_sentence(sentence: str) -> Tuple[str, str]:
  """Extract the player name and action from a sentence about sports.
  """
  ...

@traced
def sport_for(x: str)-> str:
  """The name of the sport associated with a player or action.
  """
  ...

@traced
def consistent_sports(sport1: str, sport2: str) -> bool:
  """Test if two strings describing sets of sports overlap semantically.
  """
  ...

def sports_understanding(sentence):
  """Determine if a sentence about sports is plausible or not.

  >>> sports_understanding('Santi Cazorla scored a touchdown.')
  Calling analyze_sentence('Santi Cazorla scored a touchdown.')...
  ...analyze_sentence returned ('Santi Cazorla', 'scored a touchdown.')
  Calling sport_for('Santi Cazorla')...
  ...sport_for returned 'soccer'
  Calling sport_for('scored a touchdown.')...
  ...sport_for returned 'American football and rugby'
  Calling consistent_sports('soccer', 'American football and rugby')...
  ...consistent_sports returned False
  Final answer: no
  False

  >>> sports_understanding('DeMar DeRozan was called for the goal tend.')
  Calling analyze_sentence('DeMar DeRozan was called for the goal tend.')...
  ...analyze_sentence returned ('DeMar DeRozan', 'was called for the goal tend.')
  Calling sport_for('DeMar DeRozan')...
  ...sport_for returned 'basketball'
  Calling sport_for('was called for the goal tend.')...
  ...sport_for returned 'basketball'
  Calling consistent_sports('basketball', 'basketball')...
  ...consistent_sports returned True
  Final answer: yes
  True

  """
  ...
```

Figure 3: Left, code for a mock for a simplified HB task. Right, a Program Trace prompt derived from the mock.

soning examples involve questions about bombs and shootings); that that model's RLHF instruction-training makes it reluctant to follow the prompt (the Salient Translation Errors suffered from this); or that the model's output was too long[10] (which is more frequently a problem for PTP, which has bulkier prompts and outputs.) The statistics above are computed for the subset of examples for which neither model was blocked. For Salient Translation, we added one additional sentence[11] to the prompt template used in Stage 3 to improve coverage. The number of examples blocked was moderate for every task, never more than 15%.

[10] Model output limits are trivial to get around by generating output in stages, but we leave implementation of this for later work.

[11] "Do NOT simply answer the question with a multiple-choice answer, always generate the program trace first."

B.3 Assessing type correctness

We used the `typeguard` package to verify type correctness.

To test syntactic correctness, we could usually simply call Python's `eval` function of LLM-generated text. However, in one mock, we used a library class (`intEnum`), which has a printed representation that cannot be simply read and `eval`ed. To verify correctness for this we wrote a custom parser for `intEnum` objects that de-serializes the printed representation (which looks something like `<AdjectiveCategory.COLOR: 5>`).

B.4 Are individual steps meaningful?

In the BBH suite, there were 16 different steps from five tasks with oracles. We evaluated these oracle-testable steps on the inputs that were actually used in the course of solving the dev set examples (a total of 480 oracle-testable step executions). Note that this evaluation can be expensive, since it re-

```
Consider the program fragment below.  This program fragament is
incomplete, with key parts of the implementation hidden, by
replacing them with "..." markers.

PROGRAM:
```python
{partial_program}
```

QUESTION: Predict what the output of the program above will be, given
the input shown below. Respond with the FULL program output, and ONLY
the expected program output: you will be PENALIZED if you introduce
any additional explanatory text.

```
>>> {task_name}({input_str})
```
```

```
Consider the program fragment below.  This program fragament is
incomplete, with key parts of the implementation hidden, by
replacing them with "..." markers.

PROGRAM:
```python
{partial_program}
```

QUESTION: Below is the first part of the output of the program above,
when given a new input. Respond with the remainder of the program
output. ONLY provide the remainder of the expected program output: you
will be PENALIZED if you introduce any additional explanatory text.

```
{initial_output}
```
```

Figure 4: Prompts to generate a trace and continue a partial trace.

	ours	CoT da-vinci[1]	CoT codex[2]	Chain-of-Code da-vinci [1]
Average NLP	84.0	69.7	73.5	72.6
Average Alg	88.9	74.1	74.4	94.7
Average Overall	86.4	72.0	73.9	83.2

Table 10: Overall results of Table 1 of the main paper compared with prior work: results (1) are from (Li et al., 2023), (2) are from (Suzgun et al., 2022). Prior results are over the full BBH test sets, whereas our results are on a sample of approximately 75%, as described in the text. The chain-of-code method is not a single prompt, and executes some code with Python, so is not directly comparable.

quires running an LLM many more times than does Program Trace Prompting, as there are an average of 11-12 steps per task. Table 6 shows the results, as well as macro-averaged overall accuracy.

The last column of the table also shows the accuracy of the same steps when executed in the context of problem-solving—i.e, the accuracy of the steps when they occur inside a PTP trace. Note that even the best single-step prompt is worse than performing the corresponding step in the context of problem-solving. None of the individual differences are significantly different according to the paired test used in the main paper. However, overall 10 of the 16 steps have lower average accuracy when executed independently than when executed as part of PTP problem-solving, and the remainder have the same accuracy, which seems to require explanation. It is hard to argue the set of oracle-testable steps are natural distribution, but if they were, then this difference would be significant with $p < 0.01$ with a paired t-test.

B.5 Statistical tests

The sign test was implemented by testing a binomial estimating $P(\tilde{C} = 1 | \tilde{C} \neq C)$, where C is binary correctness in the original trace and $\tilde{(C)}$ is correctness in the forced-modular trace. Non-modularity is detected if this estimate is different from 0.5 with confidence at least 95%, using `scipy.stats.binomtest` (which is an exact test).

The permutation tests uses the default settings for `scipy.stats.permutation test`, except for only 1000 samples (and a two-sided test). Jensen-Shannon divergence (which is the average KL-divergence of each sample distribution to the mean of the two distributions) was computed using `scipy.spatial.distance.jensenshannon`.

Corrections for multiple tests were done with the Bonferroni correction as implemented by `statsmodels.stats.multitest.multipletests`.

Task	Accuracy	
Step	Single-step	In PTP
tracking shuffled objs		
simulate_swap	96.7	100.0
answer_question	100.0	100.0
word sorting		
kth_letter	96.7	100.0
partition_words	100.0	100.0
sort_keys	100.0	100.0
bucket_sort	76.7	96.7
flatten	100.0	100.0
multistep arithmetic		
is_simple_expr	83.3	96.7
rewrite_expr	73.3	93.3
eval_simple_expr	100.0	100.0
eval_expr	90.0	96.7
eval_variabilized_expr	100.0	100.0
dyck languages		
is_open_paren	100.0	100.0
update_stack	80.0	100.0
e mpty_stack	86.7	100.0
boolean expressions		
solve_negation	73.3	76.7
Average	91.0	97.5
Average with ablations:		
no microtraces	87.9	
no trace parsing	86.9	

Table 11: Testing the accuracy of step-specific prompting, on 16 steps from 5 tasks that can be checked against an oracle. The "In PTP" column is accuracy of the steps executed in the process of solving task instances in the dev set; Single-step is using the single-step prompting method of Section A.3; and the remaining columns are ablations, where micro-traces and trace parsing are not used.