

Language Model Cascades

David Dohan¹ Winnie Xu¹ Aitor Lewkowycz² Jacob Austin¹ David Bieber¹ Raphael Gontijo Lopes¹
Yuhuai Wu¹ Henryk Michalewski¹ Rif A. Saurous¹ Jascha Sohl-dickstein¹ Kevin Murphy¹ Charles Sutton¹

Abstract

Prompted models have demonstrated impressive few-shot learning abilities. Repeated interactions at test-time with a single model, or the composition of multiple models together, further expands capabilities. These compositions are probabilistic models, and may be expressed in the language of graphical models with random variables whose values are complex data types such as strings. Cases with control flow and dynamic structure require techniques from probabilistic programming, which allow implementing disparate model structures and inference strategies in a unified language. We formalize several existing techniques from this perspective, including scratchpads / chain of thought, verifiers, STaR, selection-inference, and tool use. We refer to the resulting programs as *language model cascades*.

1. Introduction

Language models (LMs) have demonstrated impressive few-shot learning abilities (Brown et al., 2020; Chowdhery et al., 2022). This has led to a number of proposals to use LMs as the basis of informal reasoning, including scratchpads (Nye et al., 2021), chain of thought prompting (Wei et al., 2022; Wang et al., 2022), learned verifiers (Cobbe et al., 2021), selection-inference (Creswell et al., 2022), and bootstrapping (Zelikman et al., 2022). They have also been applied in formal mathematics settings to guide theorem provers (Polu & Sutskever, 2020). These methods involve prompting to encourage step-by-step reasoning, repeated interactions with a single LM, or multiple LMs linked together, with the models being fine-

tuned or prompted in different ways.

In this position paper, we argue that a useful unifying framework for understanding and extending this disparate body of work is in terms of probabilistic programming languages (PPL) extended to work with strings, instead of more atomic data types like integers and floats. That is, we use a PPL to define a joint probability model on string-valued random variables, parameterized using LMs, and then condition this model on string-valued observations in order to compute a posterior over string-valued unknowns, which we can then infer. We call such a probabilistic program a *language model cascade*. We show that this framework captures many recent approaches, and also allows us to tackle more complex multi-step reasoning problems. By implementing many disparate model structures and inference strategies in a single framework, we hope that language model cascades will enable the development of generic procedures to perform inference, tune parameters, and choose prompts based on end-to-end objectives.¹

2. Related work

There is a rich prior literature on probabilistic programming languages (PPLs), which extend probabilistic graphical models to support more complex joint distributions whose size and “shape” can itself be stochastic (e.g., a graph unrolled for a random number of iterations, until a data-dependent stopping criterion is met). PPLs extend traditional programming languages with the ability to *sample* from distributions and *observe* values of variables based on data (i.e. condition the model). The semantics of sample and observe vary depending on the inference algorithm. For more details, see van de Meent et al. (2018).

Recently there has been an explosion of interest in large language models, such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022). These can be used for tasks such as “zero-shot” question-answering. In this setting, we provide the question Q as a prompt to the LM, and then sample answers from the model, which we denote by $p(A|Q, \theta)$, where θ are the pre-trained model parameters. Alternatively, we can compute the MAP answer,

¹An implementation is available at [model-cascades.github.io](https://github.com/model-cascades)

¹Google Research, Mountain View, United States

²Alphabet, X, the Moonshot Factory. Correspondence to: David Dohan <ddohan@google.com>, Winnie Xu <winxiu@cs.toronto.edu>.

$$\hat{A} = \operatorname{argmax}_A p(A|Q, \theta).$$

To ensure the model “does the right thing”, we can provide a small training set of question-answer pairs, $D = \{(Q^m, A^m) : m = 1 : M\}$ pairs. This can be provided as extra context to the model, provided in the text prompt, followed by sampling from $p(A|Q, D, \theta)$. We refer to this as “few-shot prompting”. We can also fine-tune the model parameters on D to get θ' , and then sample from $p(A|Q, \theta')$.

We can improve performance by introducing an additional auxiliary “thought” variable, and then extend the model to have the form $p(A, T|Q) = p(A|T, Q)p(T|Q)$, where each conditional is computed using an LM which includes its conditioning variables as a part of its input. Work on scratchpads (Nye et al., 2021) and chain of thought (Wei et al., 2022) illustrate this, and finetune or prompt the LM to produce this auxiliary thought before answering.

We typically condition this on a small set D_S of (A^m, T^m, Q^m) triples, and optionally a larger set D_L of (A^m, Q^m) pairs. We then compute a distribution over answers to a test question using

$$\hat{p}(A|Q) = \sum_T \hat{p}(A|Q, T) \hat{p}(T|Q) \quad (1)$$

where $\hat{p}(\cdot) = p(\cdot|D_L, D_S, \theta)$ is the prior predictive distribution. (Scratchpad creates its prior predictive by finetuning, while Chain of Thought adds D_S to the LM prompt.)

In practice, we cannot sum over all possible strings T in Equation (1). The most common approach is to compute the MAP estimate $\hat{T} = \operatorname{argmax}_T \hat{p}(T)$ using beam search, and then to approximate the sum over T with this single value. More recently, Self Consistency (Wang et al., 2022) proposed to sample multiple values for T using forward sampling of (A, T) given Q , and then taking the answer A that is most common in this set².

PromptChainer (Wu et al., 2022) proposes a visual interface for composing language models together, specifying control flow and prompting strategies for each node in a chain. Nodes may query language models or external systems. Socratic models (Zeng et al., 2022) extends model chaining to the multimodal setting and demonstrates zero-shot abilities on tasks for which no single model exists.

The Eliciting Latent Knowledge proposal (ELK, 2022) suggests making latent variables explicit, modelled using a Bayesian network, to improve interpretability and safety for advanced AI systems.

Ortega et al. (2021) explains a formalism for LM finetuning with causal graphical models in order to extend the pre-

dictive capabilities of AI agents towards more adaptive behaviour. They focus on analysing an auto-regressive action (random variable) prediction scheme in the interactive setting of RL where a model is simultaneously a generator and predictor of data.

3. Cascades

In this section, we show how to create cascades of LMs to tackle various language-based reasoning problems. A *cascade* is a probabilistic program that includes string-valued random variables, sampled from an LM. For example, Figure 2 is a simple cascade for question answering. Each of the `yield` expressions return a string distributed according to the language model S .³ This program defines a joint distribution over the variables `question`, `thought`, and `answer`. Programs with complex control flow and observations are included in Appendix A.

We implement cascades as a trace-based probabilistic programming language embedded in Python via effect handlers, inspired by Bingham et al. (2018); Phan et al. (2019), and via coroutines, inspired by Kochurov et al. (2019). A pretrained LM is used to parameterize all conditional distributions. A cascade supports arbitrary control flow and recursion. While the current presentation is in terms of few-shot prompting of causal language models, we emphasize that the ideas are immediately applicable to finetuned models, masked LM setting, and other complex data types including images.

3.1. Scratchpads and Chain of thought

As our first example, we show how to represent a chain of thought (Nye et al., 2021; Wei et al., 2022) as shown in Figure 1 and subsequent graphical model figures; refer to the corresponding probabilistic programs in Appendix A. We condition the A node not just on the test question Q , but also on previous (Q^m, T^m, A^m) triples, which constitute the few-shot prompting part of the model. This is denoted by the shaded nodes inside the plate. Inference can be implemented by ancestral sampling.

3.2. Semi-supervised learning

In Section 3.1, we provided a manually created set (Q^m, T^m, A^m) triples, where the “thoughts” or “rationalizations” were provided. A more scalable approach is to define a small set D_S of such “supervised” triples, but then to provide a larger set D_L of (Q^m, A^m) pairs, which are eas-

²This bucketing is practical because most standard benchmarks have answers that are just a couple words.

³The first argument to S defines a unique name for the random variable, and the remaining arguments conditions the LM on a string prefix. A variable may be marked as observed within the program, $S('varname', \text{obs}='observed value')$, or at inference time

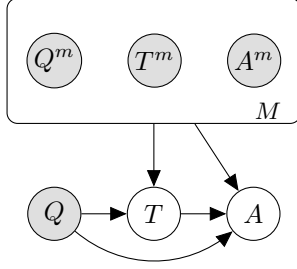


Figure 1. Question-Thought-Answer model.

```
def qta():
    q = yield S('question')
    t = yield S('thought', question=q)
    a = yield S('answer', question=q,
                thought=t)
    return a
```

 Figure 2. Chain of thought cascade in Python. Each `yield S(...)` statement samples a string from an LM. The name of the random variable is provided as the first argument to `S`.

ier to gather. We can augment the pairs in D_L by adding the hidden T^m variable to get a semi-supervised setup, shown in Figure 3.

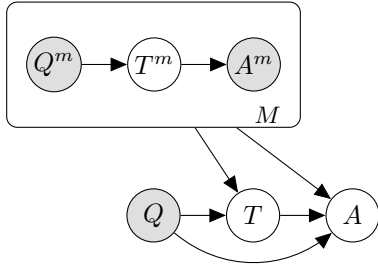


Figure 3. QTA model with hidden thoughts.

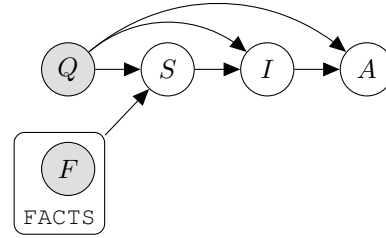
The Self-Taught Reasoner (STaR) (Zelikman et al., 2022) proposes a procedure for fine-tuning LMs in the chain-of-thought type setting. We can interpret their method as a stochastic EM-like procedure in the cascade of Figure 3. In particular, they first fine-tune on the “fully observed” dataset $D_S = \{(Q^m, T^m, A^m)\}$. Then they impute the unknown T_i values in the “partially observed” dataset $D_L = \{(Q^m, T^m = ?, A^m)\}$ during the “E” step by doing rejection sampling on $p(T, A|Q^m)$ until finding a thought which leads to the known correct answer. If sampling (T, A) given the question fails to find the correct answer, they sample thoughts from $p(T|Q^m, A^m)$. This uses a recognition network to approximately sample from the posterior distribution over thoughts given the known correct answer. They call this approach “rationale generation with rationalization”. They then update the parameters in

the “M” step based on these imputed thoughts. By interpreting the rationale generation at this higher level of abstraction, we open up the possibility of applying this tuning method to other types of cascades.

3.3. Selection-Inference

Selection Inference (Creswell et al., 2022) is a recent example of multiple interacting LM modules. It proposes splitting reasoning into: the *selection* module which selects a subset of facts given a question, and the *inference* module which infers new facts given this subset.

It may be represented by the model in Figure 4. Here S is the selection of a subset of “facts” from a pre-specified set of facts, and I is an inference driven by that fact. The S and I nodes can be iterated to do multistep reasoning. The model is “trained” by giving it examples, $D = \{(Q^m, \{F^{mj}\}, S^m, I^m, A^m) : m = 1 : M\}$, as part of the prompt.


 Figure 4. Selection inference as a cascade. Here S is the selected subset of facts and I is an inference driven by this subset.

3.4. Verifiers

Although adding explicit “thought” variables to a model has been found to improve performance, models still arrive at incorrect answers, or the correct answer for an erroneous reason. An intuitive way to improve model performance is to train it to judge whether an answer and thought are likely to be “valid”. Cobbe et al. (2021) propose using a separate model as a verifier to filter solutions to reasoning tasks.

We can create a “labeled” training set of the form $D = \{(Q^m, T^m, A^m, V^m)\}$, where we add a “verification” label $V^m \in \{0, 1\}$, representing whether the thought T^m is a valid form of reasoning for deriving A^m from Q^m , and A^m is the correct answer. This can be particularly helpful in settings where there may be more than one way of deriving the answer. The verifiers may be used to reject incorrect examples in ancestral sampling, and the thought generator may itself be conditioned on the verifiers being correct by finetuning or prompting, reminiscent of RL as inference (Levine, 2018) and goal-conditioned policies such as decision-transformer (Chen et al., 2021).

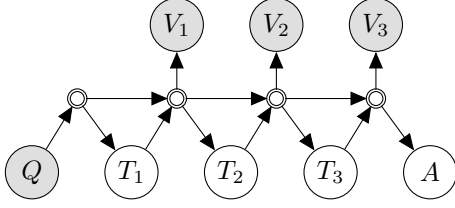


Figure 5. Verifier model. The small double-ringed nodes are deterministic buffer nodes that concatenate their inputs, accumulating all past strings. All other nodes are stochastic. The verifiers are observed to take on the “correct” value.

We can extend this to N -step reasoning as follows (where we drop conditioning on D for brevity):

$$p(A|Q, V_{1:N} = 1) \propto \sum_{T_{1:N}} p(A, T_{1:N}, V_{1:N} = 1 | Q),$$

where

$$p(A, T_{1:N}, V_{1:N} = 1 | Q) = \left[\prod_{t=1}^N p(T_t | T_{1:t-1}, Q) p(V_t = 1 | T_{1:t}, Q) \right] \times p(A | T_{1:N}, Q).$$

We can represent this as shown in Figure 5.

To see why such a verification model can be useful, consider (for simplicity) the case where $N = 1$. Suppose we have trained the model to generate valid thoughts and answers by giving it suitable training examples, and then we generate K samples $(T^k, V^k, A^k) \sim p(T, V, A | Q, D)$. We can then rank the samples for validity by computing $r^k = p(V^k = 1 | A^k, Q, D)$, and then picking the A^k with largest score r^k .

Cobbe et al. (2021) train the verifier to predict a binary correctness label. Scheurer et al. (2022) incorporates natural language feedback, and finds that learning is significantly more sample efficient. Preliminary evidence suggests that LMs are capable of critiquing their own chain of reasoning in language, in which case the verifier produces natural language and $p(V_{1:N} = 1 | Q, A, T_{1:N})$ becomes the likelihood of the verifier taking on a particular string value, such as $p(V_{1:N} = \text{”The reasoning and solution are correct.”} | \dots)$. Saunders et al. (2022) study model generated critiques in the context of summarization.

3.5. Tool-use

The applications discussed so far involve iterating a language model, within some control flow, without external feedback. There are many tasks of interest in which a model is interacting with external systems. Cobbe et al. (2021) has an LM use a calculator to solve math tasks, while Nakano et al. (2021) put an LM in a loop with a web browser to answer questions. Using PPLs to represent these

probabilistic models allows easily representing these cases, by writing the call to the external tool, such as the calculator, directly into the program. Then techniques from simulation based inference, for example, can be applied to do inference in such situations (Cranmer et al., 2020).

3.6. Twenty questions

In this section, we discuss experimental results using cascades to solve the “Twenty Questions” task from BigBench (BIG-bench collaboration, 2021). This task involves a conversation between two agents, Alice and Bob. Both agents are presented with the rules of the game, and Alice is additionally presented with a concept (e.g. ‘apple’) to describe. Bob has to guess the concept by asking a series of questions B_t of the form “Is it X?”, to which Alice answers $A_t \in \{\text{‘Yes.’}, \text{‘No.’}\}$. We repeat this process until Bob guesses correctly, or we hit the limit of T rounds. This can be thought of as a pair of interacting Markov chains, which exchange strings, until some final end state is reached, as illustrated in Figure 6.

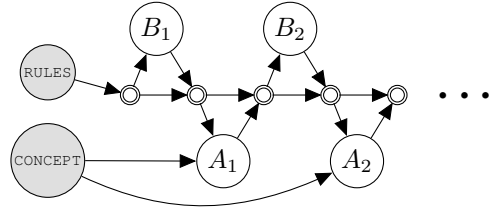


Figure 6. Twenty questions.

The goal is to infer what questions Bob should ask to guess the concept as quickly as possible. This can be cast as a reinforcement learning problem with string-valued actions, or equivalently as an inference problem where we condition on the goal state that $A_T = \text{‘yes’}$ for the soonest possible T (c.f., planning as inference (Levine, 2018)).

In our current preliminary experiments, we use a forward sampling approach (aka ancestral sampling), in which we sample 50 conversations per concept with temperature 1.0. We consider a trial successful if the target concept appears in B_t . (i.e., Bob guesses the right answer). We reject a sampling chain early if it is “malformed” (e.g., Bob generates a response that is not a question).

Bob’s turn starts with ‘Is the concept’ which we complete with the LM. Then we let Alice generate an answer; we post-process Alice’s response by replacing all mentions of the true concept with the generic word “concept”, to prevent information leakage. Using the LaMDA 137B large LM (Thoppilan et al., 2022), we find that the model is able to solve 29% of the tasks. See Appendix B for more details.

4. Discussion

We have shown how probabilistic programming provides a flexible formalism for composing models together to define complex probabilistic models over strings, placing many existing algorithms in a unified framework. While this suggests the possibility of applying a variety of existing inference and train-time techniques to the resulting models, the present work does not evaluate methods beyond rejection sampling.

We can also cast many planning and RL tasks in our framework, by using the perspective of control as inference. While we restrict presentation to the string setting, the ideas presented here are applicable to multimodal settings as well, allowing us to combine image and text models into a larger system.

A challenge applying cascades in practice is the difficulty of probabilistic inference in models with string-valued variables. Previous work in particle based inference for probabilistic programs provides some hope in this direction (Tolpin et al., 2016).

The core technical challenge is efficient inference, as is usually the case with PPLs. A key insight, which we intend to explore in future work, is that we can emulate posterior inference by training the LM to “fill in the blanks”, corresponding to the unknown variables. A similar idea is explored in foundation posteriors (Wu & Goodman, 2022), applied to Stan probabilistic programs, demonstrating that LMs are applicable to numerical data types as well. In other words, we can use LMs as proposal distributions, or guide networks. We also intend to explore fine-tuning methods, going beyond the few-shot prompting approach described here.

Recent advances in program synthesis suggest the possibility of *probabilistic program induction* (Lake et al., 2015; Goodman et al., 2014) to search for cascades which solve a target task, rather than assuming a fixed probabilistic program structure.

5. Acknowledgements

We thank Alex Gray, Andreas Stuhlmüller, Ben Poole, Du Phan, Ellen Jiang, Maarten Bosma, Matt Hoffman, Michael Terry, Sharad Vikram, Sherry Tongshuang Wu, and Tuan Anh Le for helpful discussions.

References

ARC’s first technical report: Eliciting latent knowledge - AI alignment forum, 2022.

BIG-bench collaboration. Beyond the imitation game: Measuring and extrapolating the capabilities of lan-

guage models. *In preparation*, 2021. URL <https://github.com/google/BIG-bench/>.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: Deep universal probabilistic programming. October 2018.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., and Others. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. Decision transformer: Reinforcement learning via sequence modeling. June 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Peltat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling language modeling with pathways. April 2022.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. 2021. doi: 10.48550/ARXIV.2110.14168. URL <https://arxiv.org/abs/2110.14168>.

Cranmer, K., Brehmer, J., and Louppe, G. The frontier of simulation-based inference. *Proc. Natl. Acad. Sci. U. S. A.*, 117(48):30055–30062, December 2020.

Creswell, A., Shanahan, M., and Higgins, I. Selection-Inference: Exploiting large language models for interpretable logical reasoning. May 2022.

Goodman, N. D., Tenenbaum, J. B., and Gerstenberg, T. Concepts in a probabilistic language of thought. (010), 06/2014 2014.

Kochurov, M., Carroll, C., Wiecki, T., and Lao, J. PyMC4: Exploiting coroutines for implementing a probabilistic programming framework. In *NeurIPS 2019 Workshop Program Transformations*, 2019.

- Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, December 2015.
- Levine, S. Reinforcement learning and control as probabilistic inference: Tutorial and review. May 2018.
- Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., Jiang, X., Cobbe, K., Eloundou, T., Krueger, G., Button, K., Knight, M., Chess, B., and Schulman, J. WebGPT: Browser-assisted question-answering with human feedback. December 2021.
- Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models. November 2021.
- Ortega, P. A., Kunesch, M., Delétang, G., Genewein, T., Grau-Moya, J., Veness, J., Buchli, J., Degraeve, J., Piot, B., Perolat, J., et al. Shaking the foundations: delusions in sequence models for interaction and control. *arXiv preprint arXiv:2110.10819*, 2021.
- Phan, D., Pradhan, N., and Jankowiak, M. Composable effects for flexible and accelerated probabilistic programming in NumPyro. December 2019.
- Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. September 2020.
- Saunders, W., Yeh, C., Wu, J., Bills, S., Ouyang, L., Ward, J., and Leike, J. Self-critiquing models for assisting human evaluators, 2022. URL <https://arxiv.org/abs/2206.05802>.
- Scheurer, J., Campos, J. A., Chan, J. S., Chen, A., Cho, K., and Perez, E. Training language models with language feedback. April 2022.
- Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H. S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhao, V., Zhou, Y., Chang, C.-C., Krivokon, I., Rusch, W., Pickett, M., Srinivasan, P., Man, L., Meier-Hellstern, K., Morris, M. R., Doshi, T., Santos, R. D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Agüera-Arcas, B., Cui, C., Croak, M., Chi, E., and Le, Q. LaMDA: Language models for dialog applications. January 2022. URL <https://arxiv.org/abs/2201.08239>.
- Tolpin, D., van de Meent, J., Yang, H., and Wood, F. D. Design and implementation of probabilistic programming language anglican. *CoRR*, abs/1608.05263, 2016. URL <http://arxiv.org/abs/1608.05263>.
- van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An introduction to probabilistic programming. September 2018.
- Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-Consistency improves chain of thought reasoning in language models. March 2022.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. January 2022.
- Wu, M. and Goodman, N. Foundation posteriors for approximate probabilistic inference. May 2022.
- Wu, T., Jiang, E., Donsbach, A., Gray, J., Molina, A., Terry, M., and Cai, C. J. Promptchainer: Chaining large language model prompts through visual programming, 2022. URL <https://arxiv.org/abs/2203.06566>.
- Zelikman, E., Wu, Y., and Goodman, N. D. Star: Bootstrapping reasoning with reasoning. *arXiv preprint arXiv:2203.14465*, 2022.
- Zeng, A., Wong, A., Welker, S., Choromanski, K., Tombari, F., Purohit, A., Ryoo, M., Sindhwani, V., Lee, J., Vanhoucke, V., and Florence, P. Socratic models: Composing Zero-Shot multimodal reasoning with language. April 2022.

A. Implementation

A.1. Inference

Given a program representing a probabilistic model, inference reifies specific unobserved values conditioned on observed values. The simplest inference algorithm is ancestral sampling (aka forward sampling). The basic inference API is:

```
infer(question_thought_answer_critique,
      seed=0,
      # Specify observed variables:
      observe={'question': 'Alice made 37 dollars selling ...',
              'critique': 'The reasoning and arithmetic are correct.'},
      # Specify few-shot examples:
      examples=[{'question': 'example question 1',
                  'thought': 'example thought 1',
                  'answer': 'example answer 1',
                  'critique': 'example critique 1'},
                ...])
```

A.2. Code examples

In each example below, `S` is a string distribution. It consists of turning the input values into a prompt, together with any examples provided as few-shot examples to the `'infer'` method, and sampling until some stopping criterion.

The basic question answering graph directly generates the answer given the question:

```
def question_answer():
    q = yield S('question')
    a = yield S('answer', question=q)
    return a
```

Chain of thought introduces a latent thought before producing an answer:

```
def question_thought_answer():
    q = yield S('question')
    t = yield S('thought', question=q)
    a = yield S('answer', question=q, thought=t)
    return a
```

Self critique introduces a step in which the model critiques its own reasoning in natural language:

```
def question_thought_answer_critique():
    q = yield S('question')
    t = yield S('thought', question=q)
    a = yield S('answer', question=q, thought=t)
    c = yield S('critique', question=q, thought=t, answer=a)
    return a
```

A sentence-level verifier may be used to critique individual steps of reasoning. Furthermore, when to halt generation may itself be a random variable:

```
def qta_verifier(max_steps=3):
    q = yield S('question')

    thoughts = []
    for step in range(steps):
        thought = yield S('thought', question=q, thoughts=thoughts)
```

```

thoughts.append(thought)

# Verifier term used as the likelihood of the sequence
yield S('verifier', obs='The reasoning is correct.',
        question=q, thoughts=thoughts)

# Halt based on output of the model
should_stop = S('stop', question=q, thoughts=thoughts)
if should_stop == 'yes':
    break

a = yield S('answer', question=q, thoughts=thoughts)
return answer

```

Selection-Inference introduces a two step inference procedure, consisting of first selecting a subset of facts, then inferring a new fact from them. Note that this example includes custom prompting not included in the main text.

```

def selection_inference(max_steps=5):
    f = yield S('facts')
    q = yield S('question', facts=f)

    deductions = []
    for step in range(max_steps):
        selection = yield S('selection', facts=f + deductions, question=question,
                           promptify=promptify_selector)
        inference = yield S('inference', facts=selected, question=question,
                           promptify=promptify_infer))
        deductions.append(inference)

    # Dynamic loop based on output of model.
    should_stop = S('stop', question=q, deductions=deductions)
    if should_stop == 'yes':
        break
    a = yield S('answer', question=question, deductions=deductions)
    return a

# Nodes may have custom prompts:
def prompt_selection(facts, question, selected=()):
    facts = '\n- '.join(facts)
    selected = '\n- '.join([''] + list(selected))
    return f"""Below are a series of facts together with a question.
    Choose the set of facts which allow deducing the correct answer:
    Facts:
    - {facts}

    Question: {question}

    Selected:
    {selected}"""

def prompt_inference(facts, deduction=''):
    facts = '\n- '.join(facts)
    return f"""Below are a set of facts, together with a deduction based on them:

```


Facts:
- {facts}

Therefore: {deduction}"""

B. More details on Twenty Questions

B.1. Problem definition

In this task there are two agents: Alice and Bob. Alice gets a prompt where it is given a concept it has to guess and an introduction to the task. Bob gets a prompt where it is instructed on the task. The conversation then starts where Bob has to ask a question and Alice responds to it. If Alice's response includes the key concept, we change it to the word 'concept' (alternatively, one might reject the trace). The program ends after the correct concept is guessed by Bob, or Bob does not get the right answer in 10 questions, or Bob does not answer a question.

The 40 concepts that we test the model on are: ['apple', 'television', 'dinosaur', 'airplane', 'house', 'tree', 'coat', 'shoes', 'car', 'train', 'shower', 'frisbee', 'cow', 'cosmic crisp apple', 'giganotosaurus', 'siberian huskey', 'glass micropipette', 'jog', 'catch', 'defenestrate', 'eat', 'apologize', 'operate', 'pretend', 'anger', 'love', 'hate', 'contentment', 'jealousy', 'surprise', 'disgust', 'hopefulness', 'global poverty', 'phase transition', 'positive sum game', 'beauty', 'representative democracy', 'tall', 'mauve', 'perspicacious']. The model gets right the 11 of the least abstract concepts: ['apple', 'dinosaur', 'airplane', 'house', 'tree', 'shoes', 'car', 'cow', 'eat', 'love', 'beauty'].

B.2. Examples

Alice Prompt:

X 0 Hello Alice, I am Bob.

X 1 Hello Bob

X 2 Hello Alice, we are going to play twenty questions. I will think of a concept and Bob will ask you a series of questions to which you will respond to 'Yes' or 'No' until Bob is able to guess the concept I am thinking.

X 1 Sounds good. What is the concept?

X 2 The concept is 'love'.

X 1 The concept is love ? Perfect, I got it. Bob, what is your first question? **Bob Prompt:**

X 0 Hello Alice, I am Bob.

X 1 Hello Bob, we are going to play twenty questions. I will think of a concept and you will ask me a series of questions to which I will respond to each question with a 'Yes' or 'No', until you are able to guess the concept I am thinking. What is your first question? **Common Conversation:**

X 0 Hello Alice, I am Bob.

X 1 Hello Bob, we are going to play twenty questions. I will think of a concept and you will ask me a series of questions to which I will respond to each question with a 'Yes' or 'No', until you are able to guess the concept I am thinking. What is your first question?

X 0 Is the concept a vegetable?

X 1 No

X 0 Is the concept a mineral?

X 1 No

X 0 Is the concept an animal?

X 1 No

X 0 Is the concept an instrument?

X 1 No

X 0 Is the concept an abstract concept such as love or happiness?

SUCCESS!

Alice Prompt:

X 0 Hello Alice, I am Bob.

X 1 Hello Bob

X 2 Hello Alice, we are going to play twenty questions. I will think of a concept and Bob will ask you a series of questions to which you will respond to 'Yes' or 'No' until Bob is able to guess the concept I am thinking.

X 1 Sounds good. What is the concept?

X 2 The concept is 'shoes'.

X 1 The concept is shoes ? Perfect, I got it. Bob, what is your first question?

Bob Prompt:

X 0 Hello Alice, I am Bob.

X 1 Hello Bob, we are going to play twenty questions. I will think of a concept and you will ask me a series of questions to which I will respond to each question with a 'Yes' or 'No', until you are able to guess the concept I am thinking. What is your first question?

Common Conversation:

X 0 Is the concept something that is a natural entity?

X 1 NO

X 0 Is the concept something that was invented or is man made?

X 1 the concept is concept which is something that was invented so i will answer the question with 'yes'

X 0 Is the concept a device?

X 1 Yes

X 0 Is the concept a computer?

X 1 The concept is not about a computer

X 0 Is the concept a piece of clothing?

X 1 Yes

X 0 Is the concept a shirt?

X 1 no :: the concept concept is not a shirt

X 0 Is the concept pants?

X 1 No

X 0 Is the concept shoes?

SUCCESS!

Example of wrong answer. Alice Prompt:

X 0 Hello Alice, I am Bob.

X 1 Hello Bob

X 2 Hello Alice, we are going to play twenty questions. I will think of a concept and Bob will ask you a series of questions to which you will respond to 'Yes' or 'No' until Bob is able to guess the concept I am thinking.

X 1 Sounds good. What is the concept?

X 2 The concept is 'television'.

X 1 The concept is television ? Perfect, I got it. Bob, what is your first question?

Bob Prompt:

X 0 Hello Alice, I am Bob.

X 1 Hello Bob, we are going to play twenty questions. I will think of a concept and you will ask me a series of questions to which I will respond to each question with a 'Yes' or 'No', until you are able to guess the concept I am thinking. What is your first question?

Common Conversation:

X 0 Is the concept **an animal, vegetable or mineral?**

X 1 **neither, i already know the answer, it's concept**

X 0 Is the concept **something used?**

X 1 **Yes**

X 0 Is the concept **a game?**

X 1 **Yes**

X 0 Is the concept **played with a board?**

X 1 **No**

X 0 Is the concept **from the Monopoly board?**

X 1 **No (to be alternated)**

X 0 Is the concept **from the Monopoly board?**

X 1 **No**

X 0 Is the concept **something that is kept in the box?**

X 1 **No**

X 0 Is the concept **a card?**

X 1 **No**

X 0 Is the concept **something that can be held?**

X 1 **no (concept can't be held)**

X 0 Is the concept **money?**

X 1 **No REJECT!** Bob did not get "television" in ten rounds.

B.3. Code for twenty questions

This code example demonstrates more directly constructing the prompt for a task, rather than relying on automatic prompting.

```
def twenty_questions_program(concept, max_questions):
    alice_prompt = get_prompt_from_alice(concept, max_questions)
    bob_prompt = get_prompt_from_bob(concept, max_questions)
    common_conversation = ""
    # iterate over rounds of questions and answers
    for round_number in range(1, max_questions + 1):

        current_turn = "\nX 0 Is the concept"
        # Bob's generates question. Program will be rejected if it does not generate a question.
        bob_context = bob_prompt + conversation + current_turn
        bob_response = yield S(f'bob {round_number}', prompt=prompt)
        if "?" not in bob_response:
            yield reject(reason='Bob response is not a question.')
```

```
current_turn += bob_response + "\nX 1 "

if concept.lower() in bob_response.replace('?', '').lower().split(''):
    # Bob figured it out! Score should be equal to round number.
    yield Success(num_rounds)

# Alice's turn
alice_context = get_alice_context(alice_prompt, conversation, current_turn, concept, round_number)

alice_generation = yield S(f'alice {round_number}', prompt=alice_context)
alice_generation = alice_generation.split(".")[0].split("\n")[0].split("X")[0]
# If Alice outputs the key concept, we hide it. An alternative would be to reject.
if concept.lower() in alice_generation:
    alice_generation = alice_generation.lower().replace(
        concept.lower(), "concept")

current_turn += alice_generation
conversation += current_turn

# Reject if it runs out of time.
yield reject(reason='Ran out of turns.')
```

