Overcoming the Expressivity-Efficiency Tradeoff in Program Induction

by

Sam Acquaviva

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTATION AND COGNITION

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2024

Authored by: Sam Acquaviva
Department of Electrical Engineering and Computer Science
May 10, 2024

Certified by: Yewen Pu
Senior Research Scientist, Autodesk, Thesis Supervisor

Certified by: Joshua B. Tenenbaum
Professor of Brain and Cognitive Sciences, Thesis Supervisor

Accepted by: Professor Mehrdad Jazayeri
BCS Director of Education

Overcoming the Expressivity-Efficiency Tradeoff in Program Induction

by

Sam Acquaviva

Submitted to the Department of Brain and Cognitive Sciences
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTATION AND COGNITION

ABSTRACT

People are incredibly flexible and efficient inductive reasoners. On the other hand, current approaches in program synthesis show strong domain-specific performance, but are both less sample-efficient and less flexible. Large language models improve upon this sample-efficiency and domain-generality, but lack robustness and still fall far short of people and traditional approaches on difficult induction tasks. In this thesis, we propose two hypotheses for how people seemingly overcome this trade-off between flexibility and efficiency. In the first, we propose that people may operate over an incredibly vast language which is made tractable via a strong, bottom-up proposal model. In the second, we propose that, alternatively, people may relax the necessity of such a strong proposal model by learning task-specific reasoning languages through experience. We build models operationalizing both hypotheses and show that they can improve the generality and efficiency of previous models.

Thesis supervisor: Yewen Pu
Title: Senior Research Scientist, Autodesk

Thesis supervisor: Joshua B. Tenenbaum
Title: Professor of Brain and Cognitive Sciences

Acknowledgments

Thank you to my family. Mom, Dad, Caroline, Nick, Will – I love you all. I'm so grateful to have such an incredible family, and I hope this thesis is readable.

Thank you to my friends, from MIT and home and elsewhere. Lowell, Henry, Ryan, Meagan, Samir, Jack, Jebiathan, Eamon, Chris, Snehal, Insoo – I can't wait to spend this post-graduation time with you all.

Thank you to my teammates. Running taught me how fun it can be to have big goals and go after them wholeheartedly, and you all taught me how much better it is to do it with friends. I can't imagine being at MIT without all of you. Thank you also to Mr. Hennigar and Riley for instilling in me the importance of doing something for the love of it.

Thank you to my advisors, Evan and Josh. You both have inspired me to focus on what is interesting rather than what is easy, and have always given me room to explore my ideas – regardless of feasibility or tangible progress.

Thank you to my collaborators. To all of Cocosci, but specifically to Gabe, Maddy, and Tracey for being incredibly welcoming into the lab.

And thank you, anonymous reader.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> The process of induction is the process of assuming the simplest law that can be made to harmonize with our experience. (Ludwig Wittgenstein)

The world is full of complexity. Somehow, we manage to deal with this reality. We start life experiencing it all as a "blooming, buzzing confusion" [1], and can make enough sense of the world to predict things with a great deal of certainty: I know the sun will rise tomorrow, I know if I drop my pen then it will fall, I know that if I continue writing this thesis without plugging in my laptop then the laptop will soon turn off. At the same time, I know what I *cannot* predict: I don't know if the sun will rise in 10 billion years, I don't know if the pen will bounce to the left or to the right, I don't know the exact time when my laptop will die.

How do we make these predictions?

Given a set of logical statements, one can *deduce* other statements with certainty. As the classic example goes, if I accept the premises that "all men are mortal" and "Socrates is a man", then I can *deduce* with logical certainty that "Socrates is mortal." However, many of our daily inferences are not deductions where we apply general truths to specific instances, but rather *inductions*, where we induce general truths from specific instances [2]. In the Socrates example, one could induce that "All men are mortal" from the specific facts: "Socrates is a man" and "Socrates is mortal."

The difficulty in modeling induction is that, in contrast to deduction, true premises can lead to false conclusions [3]. For example, the inductive hypothesis "The sun always rises in the east, except for on May 11, 2024 when it will rise in the west" is consistent with the experiences of all people prior to that date; however, this inductive hypothesis will be false, or at least consistently deemed unlikely by people (future readers will be able to validate that this is indeed the case).

Additionally, for any finite amount of evidence, there is an infinite number of inductive

hypotheses that are consistent with the evidence [4]. We can induce the hypothesis that the sun will rise in the east every day except for a single date in the future, for any possible future date (for which there are infinite). All of these hypotheses are fully consistent with the evidence, yet somehow we have confidence that the sun will continue to rise in the east. People can consistently make useful inductions, dealing with these infinities, every day.

1.1 A Bayesian answer to improbable inductions

We can model the probability of an inductive hypothesis H, given some evidence E, under a Bayesian framework:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{\sum_i P(E \mid H_i)P(H_i)} \tag{1.1}$$

The likelihood $P(E \mid H)$ measures how well our hypothesis explains the evidence, the prior on the hypothesis $P(H)$ measures how likely our hypothesis is absent of any particular evidence, and the marginalization $\sum_i P(E \mid H_i)P(H_i)$ is a normalizing sum over all possible hypotheses. Modeling the prior $P(H)$ corresponds to addressing the problem that inductive arguments with true premises (a high likelihood) can have an incorrect conclusion (low posterior). Figuring out how to model the marginal corresponds to addressing Hume's paradox: it is impossible to consider all possible hypotheses.

So, what priors do people have that allow them to make reasonable inductive inferences, and how do we calculate the infinite sum? Many cognitive scientists have attempted to elicit people's priors in different domains [5], and recent work in *resource rational analysis* studies how people's decisions are rational when accounting for their limited cognitive resources [6], [7]. In the case of induction, where one can only consider a small handful of possible hypotheses, these resource rational accounts try to explain strategies to efficiently approximate $P(H \mid E)$.

1.2 Program induction in artificial intelligence

Modeling induction is not just useful for explaining how people make these inferences; studying induction gives insight into how we can design machines to help us solve real problems in the world. Just as people making inductive inferences is an interesting mystery for cognitive science, designing algorithms which can make reasonable inductive inferences is an essential question for artificial intelligence.

In this thesis, we are interested in modeling a specific type of induction – *program induction*. In program induction, the "general truth" that we are trying to induce takes the form of a program. Importantly, program induction inherits the difficulties of induction more generally – there are an infinite number of programs that are consistent with a finite amount of evidence, and a program that is consistent with the evidence may not be the actual underlying program.

For example, consider the following:

$$[1, 2, 3] \rightarrow [2, 3, 4]$$
$$[8, 2] \rightarrow ?$$

Following the pattern from the first input-ouput example, what is the output for the second? It could be $[9, 3]$ if the program is *Add 1 to each number*. It could be $[2, 3, 4]$ if the program is *Output [2, 3, 4], regardless of the input*. It could even be [👽] if the program is *Output [👽] if the input is [8, 2], otherwise output [2, 3, 4]*. No matter the amount of finite evidence, there will always be an infinite number of consistent programs.

The field of *program synthesis* deals with trying to induce programs from a specification (in the previous example, the specification is an input-output example). Classically, approaches defines a formal language with explicit semantics which is searched over to find programs which satisfy the specification. Work in program synthesis typically attempts to either improve the search strategy (see Section 2.2) or the underlying language (see Section 2.3).

Recently, advances in Large Language Models (LLMs) have given rise to the study of *in-context learning*, where the LLM is given an example of a pattern in the input and must induce the correct program and apply it to a new input [8].

1.3 The trade-off between expressivity and tractability

We can further factorize Equation (1.1) to consider the language L which we are operating over:

$$P(H \mid E, L) = \frac{P(E \mid H, L)P(H \mid L)P(L)}{\sum_i P(E \mid H_i, L)P(H_i \mid L)P(L)} \tag{1.2}$$

If we are operating over natural language with each word as the basic unit, as we are (very roughly) in in-context learning, then $P(H^* \mid L)$ becomes vanishingly small (where H^* is the correct inductive hypothesis). For example, the prior probability of the natural

language hypothesis "all men are mortal" with a uniformly-weighted probabilistic context-free grammar (PCFG) over the 25,000 most common words is $1/250000^4 < 1e^{-17}$.

If the PCFG is instead uniformly weighted over the words "all", "men", "are", and "mortal", then the prior probability is $1/4^4 = 1/256$. Similarly, if the PCFG considers the 25,000 most common words but only assigns 1% of the prior distribution to the words not in the hypothesis, then the prior probability is $(4/99)^4 > 1/267$.

This simple example reveals a fundamental trade-off in program synthesis: the more expressive a language is, the more difficult induction is over that language. If a language has a large hypothesis space, then it must have an efficient inference mechanism to allow finding high posterior hypotheses: sampling without effectively conditioning on the evidence leads to very little covering of the posterior. On the other hand, a language with a smaller hypothesis space does not need rely on such a strong inference engine, but it is restricted in the space of solutions it can represent.

1.4 Two approaches to domain-general induction

This trade-off yields two distinct but complementary approaches to making tractable inductions, which will comprise the rest of this thesis.

In the first (Chapter 3), we will investigate the limits of using an expressive language (natural language [1]) with a strong proposal network (a LLM). We will provide a *rational process model* of how people induce programs, improving reported state-of-the-art on fit to human data, all while considering orders of magnitudes less hypotheses than competing approaches.

In the second (Chapter 4), we introduce a novel system for generating domain-specific languages (DSLs). These languages are significantly less expressive than natural language, but are more tractable for existing inference mechanisms. We show that this domain-general approach can outperform LLMs and existing program synthesis methods in multiple domains.

[1] We effectively consider a subset of natural language: natural language with unambiguous and explicit semantics. This is a limitation that is discussed further in Section 3.4.

Chapter 2

Background

There is plenty of work on induction, both in computational modeling and in studying how people make these inductions. Here, we review works broken up into three categories: studying and modeling human-like induction, algorithms for improved program induction by improving the inference for a given language, and algorithms for improving a language for a given inference engine.

2.1 Models of human program induction

Previous work has modeled human program induction at all three of Marr's levels of analysis [9]. At the computational level, there is an emerging literature on induction as top-down Bayesian inference over an infinite hypothesis space. These methods largely assume that this latent hypothesis space is compositional and potentially probabilistic, operating under the probabilistic Language-of-Thought hypothesis [10]–[12]. There are many other computational models for induction, such as rule-based systems [2].

Algorithmic level accounts show that people's judgements and decisions depart from these normative models, and show that sample-based approximations to the normative models can account for human errors like *anchoring* and *ordering effects* [13], [14]. These works account for how people update their hypotheses in light of new evidence, and how people update their hypotheses when given more time to think.

Recently, there has been work to provide a rational process model for human program induction using LLMs to provide natural language hypotheses [15], [16]. These works tackle concept-learning domains, where the induced program should determine if a new instance belongs to the same concept as previous instances.

Additionally, there is work finding neural correlates of these proposed sampling procedures, providing a potential implementational level account of the bases of program induction

[17], [18].

2.2 Program synthesis with strong proposal models

In the field of program synthesis, there is plenty of work improving inference within a given hypothesis space. We will define a *proposal model* as the process which generates candidate programs, potentially conditioned on the input. A proposal model aims to generate high-posterior programs. Recent work has shown incredible progress in program synthesis over general programming languages, such as Python or C, using LLMs as the proposal model [19]–[21]. When operating over smaller hypothesis spaces that are more amenable to search, other search-based methods outperform LLMs [22]. Some techniques to improve the proposals within a DSL are re-weighting a PCFG over program primitives [11], fine-tuning LLMs on DSL programs [23], re-weighting partial samples [24], using symbolic properties to re-weight the grammar [25], [26], or defining a type-system and only considering programs with valid types [27].

2.3 Library learning

Another way to improve synthesis performance is to not just learn an efficient strategy to sample or enumerate programs in the hypothesis space, but to improve the hypothesis space itself.

2.3.1 Library learning with language models

There is an emergent literature on learning libraries of reusable functions using language models. Using a simple approach of memorizing past solutions and retrieving the closest examples at test time shows improved performance [28]. More recent work focuses on not just memorizing past programs, but specifically curating libraries of reusable programs [29], [30].

2.3.2 DreamCoder, LAPS, and LILO

DreamCoder [11] is a wake-sleep algorithm that iteratively improves a DSL. DreamCoder implemented neurally-guided search (called the "Wake phase"), compression over version-spaces to find new abstractions ("Sleep: Abstraction phase"), and sampling imagined programs from the (ever-improving) DSL to train the neural-network for search ("Sleep: Dreaming phase").

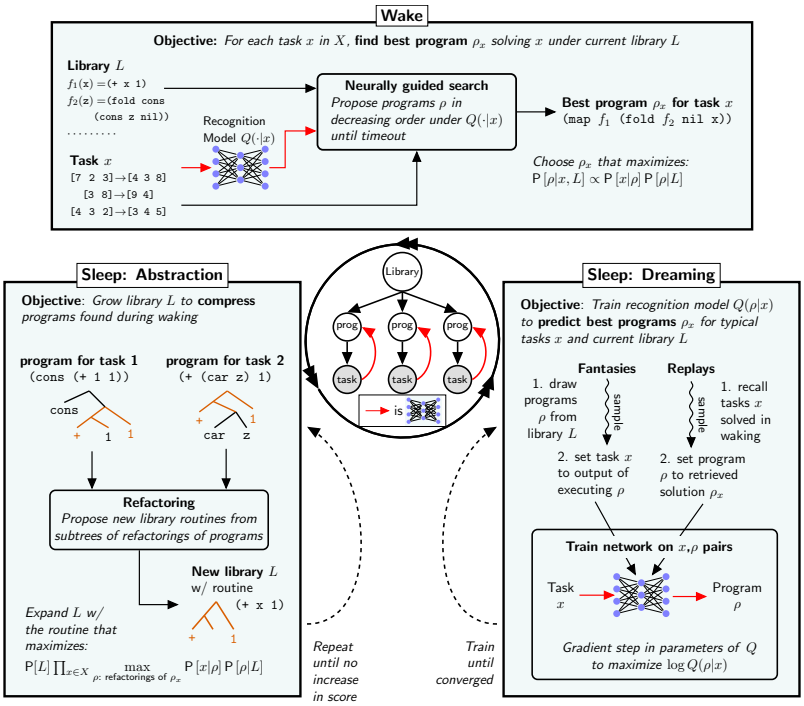

Figure 2.1: Overview of the DreamCoder architecture.

LAPS (Leveraging Language to Learn Program Abstractions and Search Heuristics) [31] extends DreamCoder to not just invert the generative process of programs, but to jointly model both language and programs. LAPS showed that incorporating language into the generative model (not just the neural search) improves the quality of the learned library, even without the presence of language at test time.

LILO (Learning Interpretable Libraries by Compressing and Documenting Code) [32] extends this spirit of using language to guide the library learning. LILO relaxes some assumptions made by the implementation of LAPS (namely, fitting translation models from scratch that rely on token-token match assumptions). Concretely, LILO uses LLMs as part of the wake phase (prompting to output solution programs directly in the DSL conditioned on examples) and the Sleep: Abstraction phase (renaming and documenting learned abstractions to help subsequent LLM inference). LILO also improves the Sleep: Abstraction phase by replacing the slow and memory-intensive version-space compression with Stitch (Top-Down Synthesis for Library Learning) [33] which is orders of magnitude more efficient.

Chapter 3

A Rational Process Model for Program Induction

How are people so efficient and flexible in their reasoning ability? Here, we present a rational process model for one hypothesis: people operate over an *incredibly* vast language which is made tractable via a strong, bottom-up proposal model. This approach continues a long line of work using sample-based Bayesian models to account for human resource rationality [13], [34]–[36]. We show that our model, which explicitly models the sequential updating of hypotheses in light of new evidence, better accounts for human accuracy and specific human errors than a sample-matched model without the sequential component.

3.1 Related work

3.1.1 Large language models

Large language models (LLMs) are inconsistent inductive reasoners. On some tasks, the ability to make inductions seems to emerge with scale [37], [38]. However, they still fall short of human performance on many abstract reasoning tasks, especially when the tasks are constructed adversarially [39]–[41].

Additionally, prior work suggests that LLMs are much better at proposing inductive hypotheses than actually applying these hypotheses. As such, using an LLM to propose inductive hypotheses in executable code and offloading execution of the rule to a code-interpreter, rather than using the LLM itself to apply the rule, often improves performance [42]. Some work shows that iteratively providing execution feedback of these induced rules to the LLM can increase performance [42], [43], while other work suggests that this may not actually help more when controlling for the number of samples [44].

3.1.2 A Bayesian model of natural language concept learning

Recent work proposes a model of human-like concept learning using natural language hypotheses generated from a LLM [15]. Modeling concept learning is very similar to modeling program induction; we can view concept learning as a special case of program induction where the input X^i is the object to be classified, and the output X^o is a binary label of whether or not the object belongs to the concept. In this setup, a hypothesis H is a rule that determines if a given input belongs to the concept in question.

Here, we reproduce the main points of the model for completeness but refer readers to [15] for a full explanation. Given a set of K input-output examples $X_{1:K}$ (where each X_k is the input-output pair (X_k^i, X_k^o)), assuming the latent concept rule H generates the examples IID, we can model the posterior over possible concept rules as

$$p(H \mid X_{1:K}) \propto p(H) \prod_{k=1}^{K} p(X_k^o \mid X_k^i, H) \tag{3.1}$$

Given this posterior, we can produce the probability of a given output X_{k+1}^o for a given input X_{k+1}^i by calculating

$$p(X_{k+1}^o \mid X_{1:K}, X_{k+1}^i) = \sum_H p(H \mid X_{1:K}) \mathbb{1}[H(X_{k+1}^i) = X_{k+1}^o] \tag{3.2}$$

where $H(X_{k+1}^i)$ is the result of applying the rule H to X_{k+1}^i. As the above sum is intractable, we construct an importance sampler q to approximate the intractable sum over H,

$$p(X_{k+1}^o \mid X_{1:K}) \approx \sum_{1 \le s \le S} w_s \text{ where } w_s = \frac{\tilde{w}_s}{\sum_{s'} \tilde{w}_{s'}} \text{ and } \tilde{w}_s = \frac{p(H_s) p(X_{1:K}^o \mid X_{1:K}^i, H_s)}{q(H_s \mid X_{1:k})} \tag{3.3}$$

In the model from [15], they draw S samples from an LLM and approximate the importance weighting by de-duplicating hypotheses, so $q(H_s \mid X_{1:k})$ is approximated as a uniform distribution over unique hypotheses. The likelihood $p(X_{1:K}^o \mid X_{1:K}^i, H_s)$ is task-dependent, but always translates each natural language hypothesis H_s to a code hypothesis (using another LLM call) which is then executed. The prior $p(H_s)$ comes from training a linear model on text features extracted from H_s to fit human data, which leads to notable improvement in fit to human data over a fixed prior.

3.1.3 Sequential Monte Carlo

When the data is presented sequentially, as it is in our setup, it is natural to use Sequential Monte Carlo (SMC) to approximate the posterior [45]. In SMC, a finite set of particles are continually adjusted to approximate the posterior. Importantly, SMC does not require global information about the posterior over these particles, but only *local* information.

One implementation of SMC is to iteratively *rejuvenate* and *filter* particles. When new evidence comes in, local mutations to the particles with the highest importance weights w are sampled (rejuvenation). Then, the pool of sampled mutations and original particles are re-sampled with probability proportional to their importance weights, to the set number of particles (filtering).

Across domains, people demonstrate anchoring and bias effects that are well captured by SMC [14]. However, these accounts usually consider orders of magnitudes more particles than is likely possible for a person to consider. In contrast, concurrent work shows that it is possible to model active inference in concept learning as SMC over natural language hypotheses with a plausible number of hypotheses [16].

3.2 Model

3.2.1 Domain

We focus on the List Functions domain, where the task is to, given a set of input-output examples, induce a program which maps each input list to its corresponding output list. Then, the learner must apply this induced program to a new input for which the output is hidden. One such task from the domain is:

$$[3, 1, 9, 0, 7] \rightarrow [1, 9, 0]$$
$$[2, 1, 3, 4, 6, 9] \rightarrow [1, 3]$$
$$[1, 5, 4, 2, 8, 3, 0, 6] \rightarrow [5]$$
$$[4, 1, 2, 3, 5, 0, 7, 6, 9, 8] \rightarrow ?$$

In the example above, the program is *Take the first N items after the first item, where N is the value of the first item.* We use data from [12] which collected human performance from 389 participants on a corpus of 250 such tasks. In this domain, the tasks are presented iteratively: a participant is shown each example one at a time and asked to predict the output for a new list. So, we have *learning curves* for participants across the number of examples shown for each task. For simplicity and to align with previous work, we restrict our analysis to the first 100 tasks of the corpus.

3.2.2 Overview

In contrast to [15], and in line with concurrent work [16], we don't just use an importance sampler to approximate the posterior for each example separately. Instead, we approximate the sum in 3.2 using Sequential Monte Carlo, with rounds of rejuvenation and filtering.

In the first iteration (when shown the first input-output example), we sample S hypotheses $H_1, H_2, ...H_S$ from the proposal distribution q and calculate the importance weight $w_H^1 = \frac{p(H)p(X_1^o|X_1^i,H)}{q(H|X_1)}$ for each. In subsequent iterations (when shown more input-output examples), we incorporate the new example's likelihood by re-weighting each hypothesis: $w_H^{k+1} = w_H^k p(X_{K+1}^o \mid H, X_{K+1}^i)$. Then, during the rejuvenation step, we sample N mutations for each of the top M particles from $q(H_{K+1} \mid H_K, X_{1:K})$. Then, we re-sample S hypotheses from this pool of $S + M \cdot N$ hypotheses with their sampling probability proportional to their importance weights.

3.2.3 Likelihoods, priors, and proposals

In our experiments, we use gpt-4-0613 [21] as our proposal distribution q over natural language language hypotheses H. Preliminary experiments showed that other publicly-accessible LLMs are much worse at induction tasks. For our prior, we find that a simple length-prior $p(H) \propto \frac{1}{|H|}$ works just as well as a linear prior model fitted to human data, in contrast to previous work in concept learning.

For our likelihood function, we use a simple likelihood function

$$P(X_{1:K} \mid H) = \prod_{1 \leq k \leq K} (1-\theta)\frac{\mathbb{1}[X_k^o = H(X_k^i)]}{K} + \theta\frac{\mathbb{1}[X_k^o \neq H(X_k^i)]}{K} \tag{3.4}$$

where θ is the probability that an example is mislabeled. Note that the likelihood is monotonically increasing with the number of correct examples; a hypothesis will always have a higher likelihood if it can directly explain more of the data than another hypothesis. Initially, we set $\theta = \frac{1}{100}$, and after proposing all hypotheses, we fit θ to human data using k-fold cross validation, with $k = 10$. Fitting the likelihood parameters, in this case just θ, while running the model is interesting future work that may improve performance.

3.3 Results

We compare our model – **Full model** – to two baselines. The **Hacker-like (HL)** model [12] is a search algorithm over *meta-programs* which uses *term-rewriting systems* to drastically improve search efficiency over alternative symbolic search algorithms like Fleet [46] and

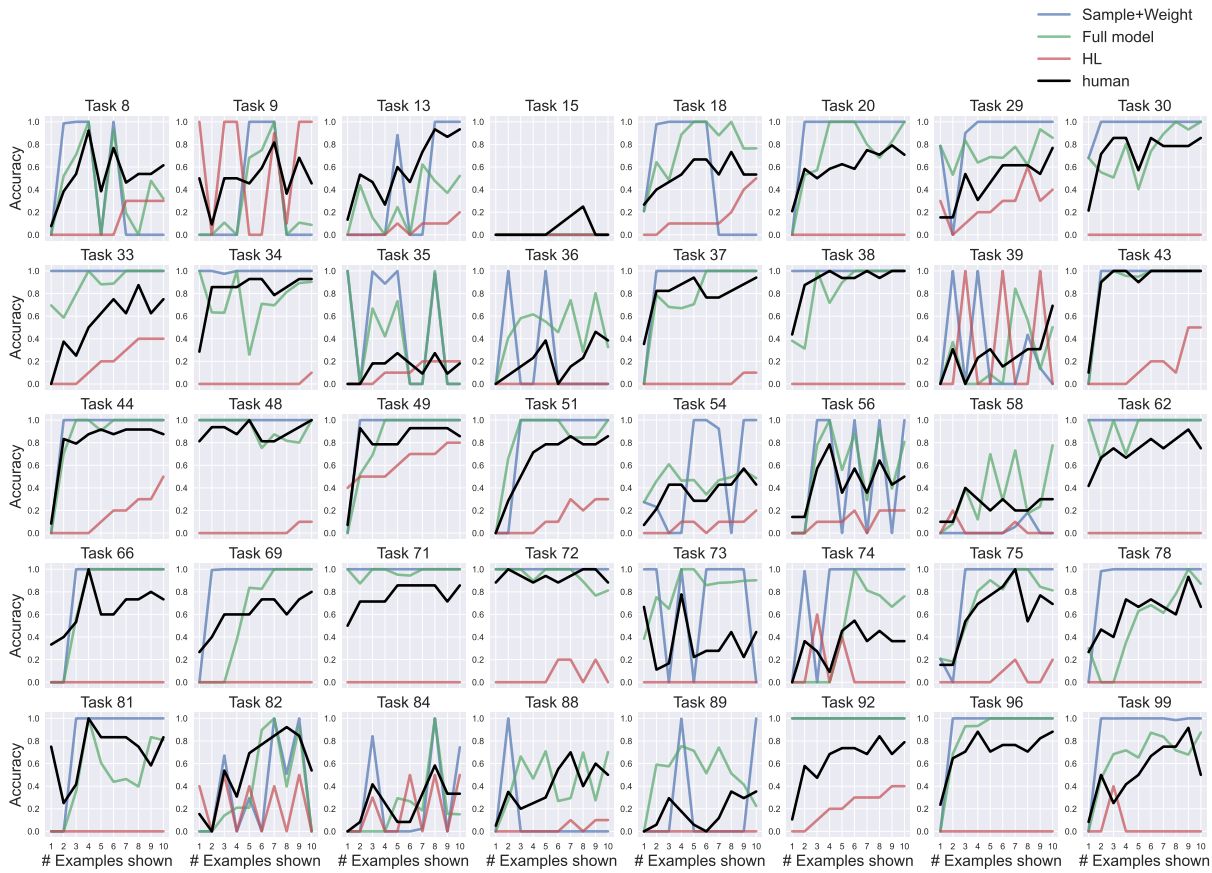

Figure 3.1: Model accuracy and human accuracy for a random subset of 40 tasks, by example. ($s = 5$)

Metagol [47]. The **Sample+Weight** baseline removes the sequential aspect of our model, treating each example k as a separate task. This is essentially Equation 3.3, and can be viewed as a sample-matched ensemble baseline. For each baseline, we will also specify the sample budget (in the case of Sample+Weight and the full model) or the search budget (in the case of HL).

3.3.1 Fit to human accuracy

In Figure 3.1, we plot sample-matched model accuracy versus human accuracy for the **Full model**, **HL**, and **Sample+Weight** for a random subset of 40 tasks. Each model uses 5 samples / search steps per example. Across the 100 tasks, the **Full model** has the closest fit to human data (as measured by mean-squared error (MSE) across the 10 examples), for 70 of 100 tasks, **Sample+Weight** for 17 of 100 tasks, and **HL** for the remaining 13. In Appendix A.2, we plot all 100 tasks with these models and **HL** with 500k search steps, as well as provide the MSE for each model for each task.

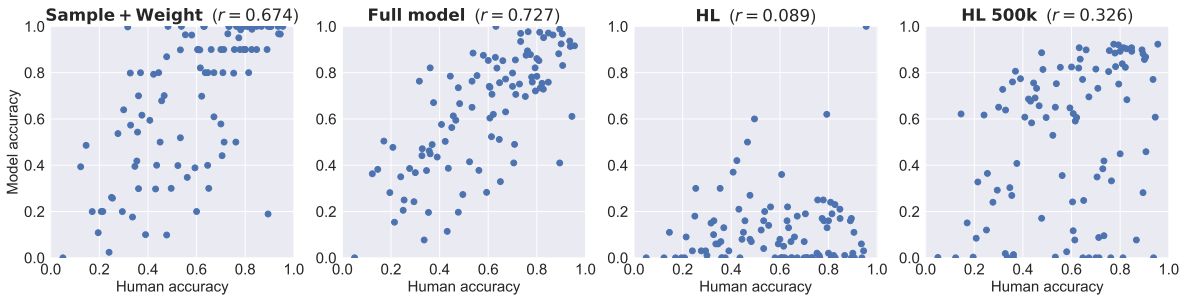

Figure 3.2: Model mean accuracy across examples versus mean human accuracy across participants and examples ($s = 5$ except for HL 500k, where $s = 500,000$).

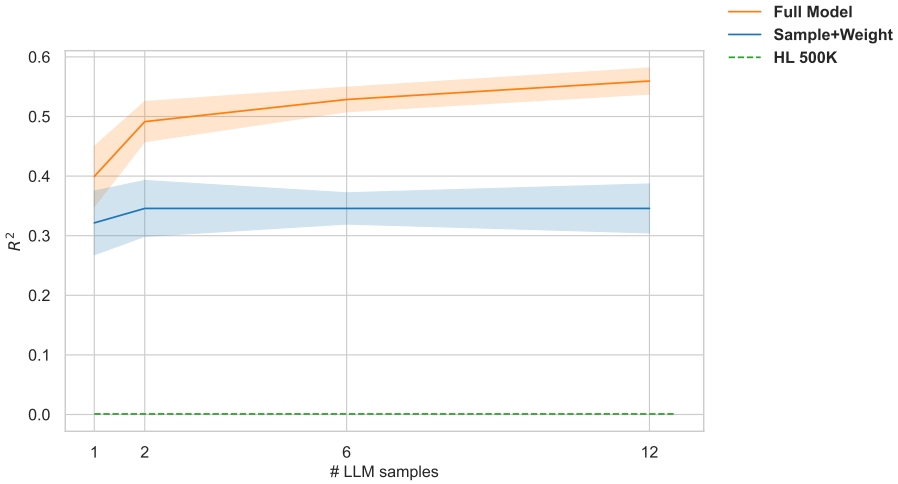

Figure 3.3: Fit to human judgements r^2 across different sample budgets.

In Figure 3.2, we plot model average accuracy against the human average accuracy across all 100 tasks. In this plot, the accuracy is averaged across examples. For each model, the accuracy is weighted by the posterior: if the model has one correct hypothesis with a posterior of 0.8, the accuracy for that example would be 0.8.

In Figure 3.3, we show the fit to human data (as measured by r^2) as a function of the number of hypotheses considered (shaded regions represents standard deviation across 3 runs). Due to computational restrictions, we limit the experiments in this plot to a difficult subset of 20 tasks and to the first 6 examples, where most of the learning happens (see Appendix A.3 for information on selecting this subset). Note that **HL**, with 500k search steps, has near-zero correlation on this difficult subset. We show that, across sample budgets, our model's performance better captures the average human performance across tasks better than sample-matched **HL** and **Sample+Weight**.

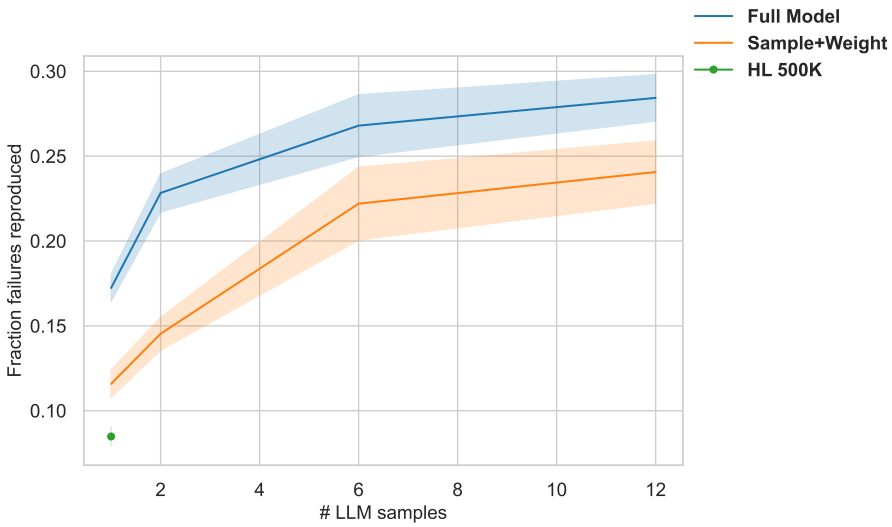

Figure 3.4: Fraction of human errors reproduced by the model.

3.3.2 Fit to human errors

When modelling human program induction, we are also interested in comparing the *errors* of the model and to human errors. On many induction tasks, both program synthesis models and LLMs err in qualitatively different ways than people [41].

Analyzing the errors in the List Functions domain is difficult in two ways: we do not explicitly elicit rules from participants so we cannot directly compare induced rules, and we do not give an option for "I don't know", leading to many participants outputting random lists that are not representative of good-faith attempts, but rather just to move on to another example. We leave collecting more data that addresses these limitations as future work.

With these limitations in mind, we restrict our error analysis to explaining errors after a successful response. Concretely, we measure, for each example, the overlap between the model's responses and the set of human incorrect responses after a correct response. These errors are shown in Figure 3.4 (shaded regions represents standard deviation across 3 runs). As **HL** does not maintain a fixed set of s hypotheses during search, we only plot the error analysis for 1 sample.

We find that explicitly modeling the sequential nature of hypothesis formation, as we do in the **Full model**, improves coverage of human errors.

3.4 Discussion

We present a rational process model for how people update their hypotheses in light of new evidence with a realistic number of hypotheses considered, integrating classic models of hypothesis adaptation with LLMs to improve fit to human data. However, there are clear limitations of the model, which we will outline here and leave for future work.

Adapting hypothesis in time

There is plenty of work modeling how people update their hypotheses when given more time to think, whereas we only model how people update their hypotheses when given new evidence.

Specifically, at time t, the n samples our model generates are IID. On the other hand, people's hypothesis generation process is much richer: people notice properties of the input, maintain partial hypotheses, and let their current hypotheses guide their next hypothesis. However, our experiments revealed that, contrary to prior work, it is difficult to improve induction performance beyond just sampling more IID hypotheses.

Concretely, we tried three main techniques to model how people might update their hypotheses in time. In the first, and most straight-forward, we tried two standard prompting techniques purported to "improve reasoning" – Chain-of-Thought [48] and Tree-of-thoughts [49]. However, we found that neither of these improve performance when controlling for total number of samples.

We also tried using *post-hoc attention steering* [50], where we can up-weight attention to specific parts of the input in specific attention heads. We searched over manually-specified different parts of the input to attend to (e.g., the first input-output examples, the first element of each input list, odd-numbered indices, etc...). In preliminary experiments, we found that attention steering was ineffective to usefully guide the rule inductions in Llama 2-13b [51] – the largest open source model we had the computational resources to test. However, there are two reasons to believe that this approach may work if scaled up to larger models: program induction is a difficult task that does not seem to emerge until larger scales [38], and the specific attention heads that are used to steer the attention were identified on non-induction tasks and may be task-specific.

Finally, we tried using *execution feedback*, where we include the results of the execution of incorrect hypotheses in the LLM prompt. Contrary to prior work on using LLMs for program induction [42], and in line with recent work analyzing the use of LLMs for program synthesis [44], we find that providing execution feedback provides no benefit over IID sampling the same number of hypotheses.

Human studies

As noted in Section 3.3.2, one limitation of our error analysis is that it is unclear whether people's incorrect answers are reflective of a good-faith attempted answer, or a random response because they could not determine the pattern. Additionally, we do not have people's explicit hypotheses for what they believe the program is. One could alleviate these issues by running a study which includes an "I don't know" option, asks for confidence scores after each response, and asks participants to describe what they think the program is in natural language after each example.

Additionally, because most program induction domains do not collect human data, we can only test our model's accuracy on these domains rather than its ability to explain human behavior. Collecting more human data in more program induction domains would provide a rich problem set for future modeling work.

Moving beyond procedural, deterministic rules

Our method, as is, relies on translating natural language hypotheses into deterministic, executable code. However, this severely limits the expressibility of the reachable hypothesis class. For instance, if the program is to identify objects in a list that "are more pointy than round" or "tree-like", it will be very difficult to write a Python program that can solve this task. On the other hand, LLMs are imbued with enough training data to accurately make these kinds of judgements [52].

One benefit of the Bayesian approach is that it is very natural to introduce these non-procedural, non-deterministic rules. Similarly, one can incorporate our model with other perceptual models to move beyond textual inputs.

Chapter 4

Bootstrapping a DSL from Scratch

An alternative hypothesis to the idea that people operate over a single expressive language (as modeled in Chapter 3) is that people learn constrained, task-specific languages. In this section, we present a library learning system inspired by this alternative hypothesis: instead of using a general and expressive language with a very strong proposal model, we induce a rich, domain-specific language that can perform induction within a domain with the simplest possible proposal model.

4.1 Methods

4.1.1 Typed enumeration

We assume each task has a given type. For example, every task in the list-functions domain has type $int \to int$. In contrast to Chapter 3, where we *sampled* programs from the language model in the form of natural-language hypotheses, we *enumerate* over validly-typed programs [27] in order of prior probability until a given timeout.

4.1.2 DreamCoder and LILO

Our approach builds on DreamCoder (see 2.3.2). This chapter is interested in the limits of inducing programs by improving the underlying language alone as opposed to improving performance by using a strong proposal model. As such, for all models across experiments, we will not use neurally-guided search (as in DreamCoder) or sample programs in the DSL from the LLM (as in LILO). Introducing these task-specific inference methods would undoubtedly improve performance for all baselines and the proposed model, and we leave this area for future work.

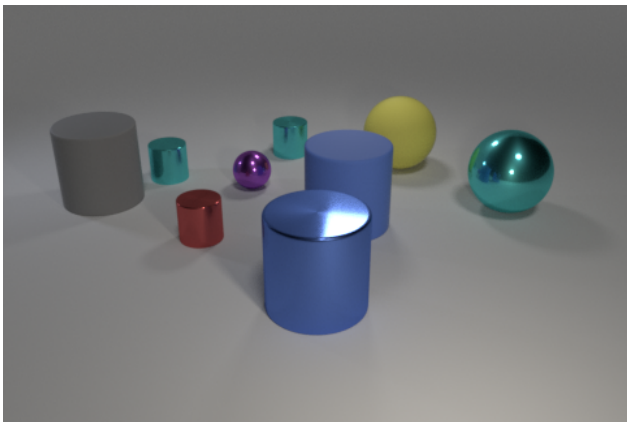

Figure 4.1: Example CLEVR scene.

4.1.3 Domains

We conduct experiments in three domains: List Functions, Regular Expressions, and Compositional Language and Elementary Visual Reasoning (CLEVR). The List Functions domain is described in 3.2.1. The Regular Expressions domain [53] consists of string transformations. One such task from the domain is:

$$\text{abstinent} \rightarrow \text{ubbstubnubnt}$$
$$\text{bucklers} \rightarrow \text{bubcklubrs}$$
$$\text{melodiousness} \rightarrow \text{mublubdubububsnubss}$$

In the example above, the program is *Replace each vowel with "ub"*. The CLEVR domain [54] consists of questions about scenes of objects, designed originally to test visual question-answering models. One example task from CLEVR is to answer the question: "Are there an equal number of large things and metal spheres?" for a particular set of objects, such as those in Figure 4.1.

For all experiments, we strip the question from the input, creating the difficult induction problem of inducing the question from only input-output examples. Additionally, since we are interested in program induction rather than visual scene understanding *per se*, we codify each image as a list of object properties and their relative positions, as in [11].

4.2 Model

4.2.1 Overview

We view our method as a natural combination of DreamCoder / LILO (see Section 2.3.2) and the inductive ability of LLMs (see 3.1.1). In each iteration of DreamCoder, new solu-

tions are found in the underlying DSL via *search* which bootstraps *compression* (because there are more solutions to compress) which in turn bootstraps the next iteration of search because the library is better tuned to the domain. Importantly, DreamCoder relies on an initial DSL which is expressive enough to represent all relevant problems in the domain and compact enough so that search is tractable (although this limitation is somewhat mitigated by neurally-guided search and, in later iterations, compression). On the other hand, LLMs can demonstrate strong inductive reasoning abilities, but are brittle, especially in adversarial domains.

We propose a method to incorporate LLMs into the virtuous cycle of bootstrapping as demonstrated in DreamCoder. Specifically, we

1. Sample programs from a LLM to solve tasks in a general programming language.
2. Translate the solution programs into declarations of primitive functions.
3. Combine primitives from all solutions into a single library and compress the library.
4. Explicitly search over the library.

In this way, we maintain the benefit of explicit, enumerative search without requiring manual design of a domain-specific language. Note that every step of the process can be parallelized across tasks.

Solving individual tasks

Initially, we begin with a set of unsolved, related tasks $\{T_1, T_2, ..., T_n\}$. In the first iteration, we prompt the LLM to solve each task T_i in Python. Importantly, we can check whether or not a sample from an LLM can solve a given task by executing the Python code on a set of hidden examples. As such, we then have a subset of solved tasks. We find that, in the three domains we test, the LLM can solve a non-zero amount of tasks with a moderate number of samples, beginning the bootstrapping process without any human labelling. However, in the case where an LLM cannot solve *any* tasks in a domain, the process could be bootstrapped by providing either solution programs directly or descriptions of the solution.

In subsequent iterations, we can improve LLM induction performance using the set of already-solved tasks and the current library. In our experiments, we try a straightforward technique: selectively choosing few-shot examples from the set of solutions.

For all the experiments in this thesis, we use gpt-3.5-turbo-0125 to solve tasks.

Translate solutions

In order to search over a language, we need to generate a language to search over. Since our search strategy is defined over typed, functional programs (see Section 4.1.1), we need to convert our Python programs into a set of typed primitives that can be composed to form a functional solution to the original problem.

This is a difficult problem, as it requires not just translating an imperative program into a functional program, but also finding a reasonable granularity of decomposition. For example, any imperative function could be translated into a functional program by defining the entire program as a single primitive; however, this is too specific a primitive to be generally useful. On the other hand, we can decompose the program into byte-code instructions, but this decomposition would yield far too many primitives to be useful for search.

In lieu of a principled, effective technique for automatic decomposition into primitives, we find that LLMs are quite strong at finding decent decompositions of functions. So, we use another LLM call to translate solution programs into lists of typed primitives. Importantly, once again, we can check if these primitives can solve the original task. In contrast to the previous step, where we can just execute the sampled code, we must search over these primitives to determine if some composition of the primitives can indeed solve the task.

For all the experiments in this thesis, we use gpt-4-turbo-2024-05-13 to translate tasks into lists of primitives.

Compress primitive sets

From the last step, we have a set of primitives for each task. However, in order to search, we need a single library of primitives. As we are attempting to build a DSL from separate, imperfect sets of primitive – in contrast to DreamCoder which operates over a minimal, hand-designed DSL – we have a unique problem of *primitive redundancy* which necessitates an additional step to effectively compress our task-specific primitives into a single library.

In the simplest case, we can re-use observationally-equivalent primitives in different tasks. For example, if two different tasks use the primitive *map*, then we should only define one *map* primitive. In another case, we may have a primitive that can be constructed by a composition of other primitives. For example, the primitive *add_1(x) = x + 1* can be constructed by the primitives *add(x, y) = x + y* and *1* (*add(x, 1)*).

We address both of these cases with a simple procedure: search for replacements to each primitive. Once this is complete, we then use the compression technique Stitch [33] to add compositions of primitives to our library.

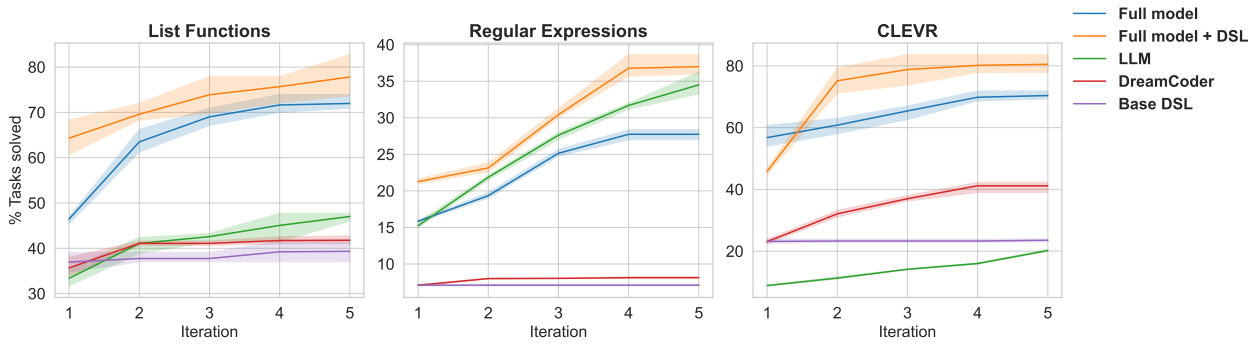

Figure 4.2: Online synthesis performance for baseline models.

Search for solutions

Finally, we use this library – with each primitive weighted via the inside-outside algorithm over previously-found solution programs – to search for new solutions in the domain using the type-constrained search outlined in Section 4.1.1. As noted earlier, this search is not conditioned on the input.

4.3 Results

We compare our model – **Full model** – to three baselines. **LLM**, which adds each LLM solution to the library as a single primitive; this can be viewed as an upper bound on the performance of the LLM. **Base DSL** is the DSL used to start the DreamCoder loop in the original DreamCoder paper, and **DreamCoder** is the DreamCoder model referenced throughout the thesis (but without neural search). Additionally, we plot **Full model + DSL**, which uses our method to add primitives to the library, but starts the library with all of the primitives used by the **DreamCoder** baseline. Each model uses 8 LLM sampler per task per iteration and 100k search steps in the DSL, and we use a random subset of 100 tasks per domain.

4.3.1 Synthesis performance

In Figure 4.2, we plot the performance of each model across 5 iterations (shaded regions represent standard deviation across 3 runs). Note that, in all three domains, the bootstrapped DSL can solve tasks not originally solved by the LLM. Additionally, initializing the DSL with the base DreamCoder primitives further improves performance, and shows that the induced library can improve upon existing libraries. As mentioned in Section 4.1.2, it will also be a useful comparison in future work to compare against DreamCoder with neurally-guided

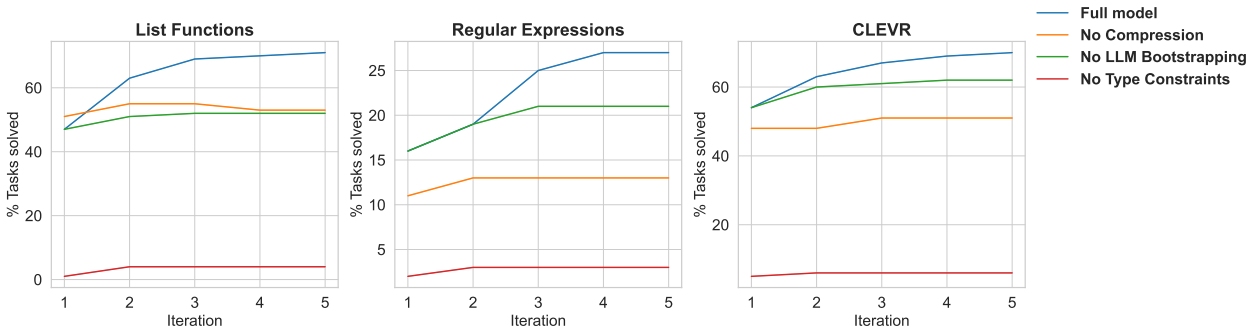

Figure 4.3: Online synthesis performance for ablations.

search and LILO with LLM search.

In Figure 4.3, we plot the performance of ablations to the model across 5 iterations. The **No Compression** ablation removes the primitive replacement and Stitch compression step. The **No LLM Bootstrapping** baseline removes bootstrapping the LLM with few-shot solutions from previous iterations. The **No Type Constraints** baseline performs search over the DSL, but removes the type-constraints from the search.

4.3.2 Qualitative library analysis

We also analyze the generality of the primitives in the induced library. In Table 4.1, we show the empirical mean for the number of solved tasks each primitive appears in, for the **Full model**, **DreamCoder**, and **No Compression** ablation. Note that reusable, generally useful primitives should appear in many tasks.

Model	**List Functions**	**Regular Expressions**	**CLEVR**
Full Model	0.285	0.299	0.604
DreamCoder	0.305	0.600	0.253
No Compression	0.071	0.078	0.091

Table 4.1: Mean fraction of solved tasks each primitive appears in.

In Table 4.2, we highlight the most useful primitives (as measured by number of tasks they appear in) from the induced library in each domain. We put the full induced libraries (including primitive types and function declarations) for each domain in Appendix B.2.

4.3.3 Bootstrapping

In all of the above experiments, we bootstrapped the LLM with few-shot examples from the previous solutions. However, is it the *search* solutions that are bootstrapping the LLM or

List Functions	Regular Expressions	CLEVR
length	length	==
-	slice	if
tail	concat	get
int → list	subtract	any
<	if	filter

Table 4.2: Most useful primitives by domain.

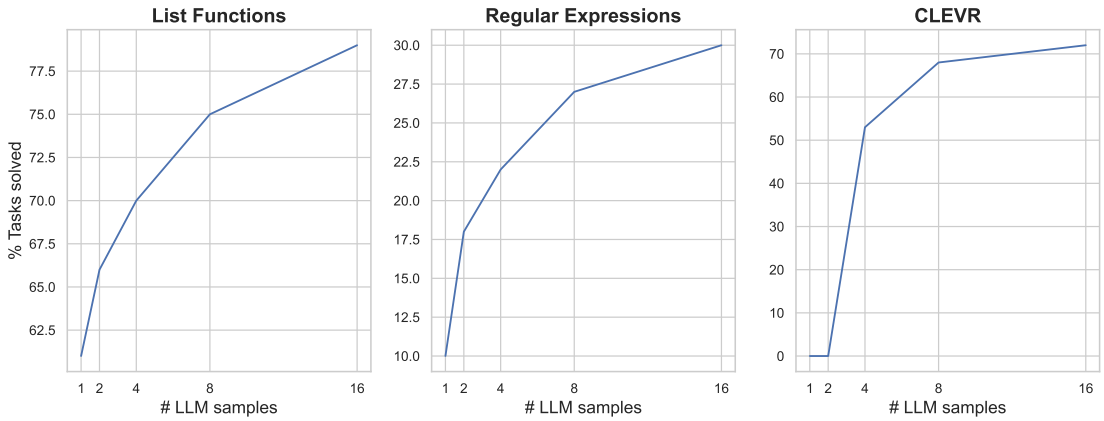

Figure 4.4: Search performance as a function of LLM samples.

the solutions that the LLM solved in previous examples?

We find that while bootstrapping the LLM with in-context examples *does* improve sample-efficiency, increasing the size of this pool of candidate examples with solutions found via search *does not* further increase performance. We try three example selection strategies: random, embedding distance of input-output examples, and string-edit distance of the in-context example function applied to the same function. We find that performance is comparable for all three strategies, and is not improved when adding search solutions to the candidates. Improving example selection, and trying different ways of bootstrapping the LLM (such as including the library of functions, for example) are future work.

On the other hand, we do find that increasing the number of samples of the language model *does* improve search performance, as shown in Figure 4.4.

4.4 Discussion

We present a model of bootstrapping a DSL from scratch and demonstrate the quality of the learned library across domains. This method combines the domain-generality of LLMs with search to improve program induction performance within a domain. There are many clear extensions to this work, some of which we will outline here.

Scaling up experiments

In this thesis, which is concerned with modeling how people can do program induction with modest computational budgets, we purposely restricted the search budgets of our method and baselines. However, much of the performance of models like DreamCoder and LILO only come with very large search budgets ($>$ 100x the budget used in our experiments). Scaling up our experiments and comparing would perhaps allow for "fairer" comparisons with these methods, in addition to potentially unlocking improved performance.

We restricted our analyses to a 100-task subset in each domain. Including more tasks per domain would provide a higher density of relevant functions in the domain, and could yield further improvements to the induced libraries.

Additionally, we did not consider strong proposal models. One benefit of maintaining a constrained, formal language as we do in this paper (and as is done in DreamCoder and LILO) is that you can sample programs from the DSL to train a recognition model to re-weight the library (conditioned on a task) to make search more efficient. Additionally, LILO showed that using the LLM to sample programs directly in the DSL improves synthesis performance and library quality. We leave integrating recognition models and LLMs for DSL-inference as future work.

Moving beyond functional programs

The lambda-calculus we are searching over only represents functional programs. On the other hand, programmers typically operate in imperative-style languages. This difference may limit the utility of the learned libraries as tools for people or language models.

Moving beyond deterministic input-output examples

Throughout the thesis, we only consider program induction in domains with deterministic input-output examples. However, within the Bayesian framework we present, it is straightforward to frame probabilistic tasks and other formats. However, this presents interesting challenges for both inference and compression, which we leave as future work.

Task-specific libraries

In this work, we ignore incorrect programs sampled from the LLM. However, these programs may have some utility: capturing some aspect of the correct program even if they are incorrect. Future work could try using primitives derived from partially-correct programs in addition to the base DSL to solve solutions in a "task-specific library".

Chapter 5

Conclusion

We highlight a fundamental trade-off in program induction between the *generality* of a language and the *tractability* of inference in that language. Somehow, people overcome this tension with demonstrated flexibility and efficiency in problem solving. We propose two hypotheses to explain people's performance and present two models inspired by these hypotheses. In the first, the model operates over an incredibly general language, and uses a strong proposal model to make inference tractable. In the second, the model operates over a tractable and domain-specific language that is tractable even with the simplest of proposal models.

Looking forward, in addition to the straight-forward extensions mentioned in the conclusions of the previous chapters, we want to combine the strengths of these models and validate them with human experiments. We believe that the emphases of both models – the proposal model and the underlying language – are important aspects of the recipe that comprises human inductive ability.

We hope that we can build a model that can, in fact, induce that the sun will rise tomorrow.

Appendix A

Program Induction Rational Process Model Supplement

A.1 Prompts

Here, we list all of the prompt templates used to generate, mutate, and translate natural language hypotheses. Note that these prompts are incredibly minimal, as the motivation of the project was not to maximize performance or perfectly tune prompts, but rather to provide a framework for modeling human-like program induction. Future work here would likely yield further improvements.

A.1.1 Hypothesis proposal prompt

```
Generate a rule that maps the following inputs to their corresponding
   outputs.

{examples, in the format [a, b, c] -> [x, y, z]}

Please format your rule as follows:

Rule: <Your rule>
```

A.1.2 Hypothesis translation prompt

```
    You are an expert Python programmer. Write a Python function `fn`
       for the following rule. The input is a list of integers. The
       output is also a list of integers.

Rule: {rule}
```

A.1.3 Hypothesis mutation prompt

```
Your rule: {rule}

This rule does not work for the following examples.

{list of examples where the executed rule is incorrect, with the input,
    predicted output, and correct output shown}

Generate a new rule that maps the given inputs to their corresponding
   outputs. Respond in the following format EXACTLY:

Rule: <Your new rule>
```

A.2 Full model prediction

In Figure A.1, we plot human accuracy and model accuracy for the models **Sample+Weight** with $s = 5$, **Full model** with $s = 5$, and **HL** with $s = 5$ and $s = 500k$.

In Table A.1, we show the difference between the model prediction and human error (as measured by mean-squared error) for the same models as above. The best fit is in **bold** and the second-best fit is **italicized**.

Table A.1: MSE per task for each model.

Task	Sample+Weight ($s = 5$)	Full model ($s = 5$)	HL ($s = 5$)	HL ($s = 500K$)
c001	0.127	**0.054**	0.558	*0.06*
c002	0.259	*0.252*	0.376	**0.223**
c003	0.154	**0.007**	0.424	*0.019*
c004	0.336	**0.176**	0.216	*0.194*
c005	0.061	**0.045**	0.052	*0.051*
c006	*0.065*	0.119	0.404	**0.056**

Task	Sample+Weight ($s = 5$)	Full model ($s = 5$)	HL ($s = 5$)	HL ($s = 500K$)
c007	0.174	**0.044**	*0.125*	0.209
c008	0.196	**0.069**	0.227	*0.118*
c009	0.206	*0.137*	0.179	**0.12**
c010	0.194	**0.094**	*0.111*	0.133
c011	*0.063*	0.075	0.453	**0.024**
c012	0.186	*0.069*	**0.038**	0.079
c013	0.145	*0.118*	0.328	**0.072**
c014	0.221	*0.119*	**0.032**	0.126
c015	*0.01*	*0.01*	*0.01*	**0.009**
c016	0.226	**0.021**	0.409	*0.032*
c017	0.152	*0.083*	0.278	**0.071**
c018	0.245	**0.071**	0.169	**0.071**
c019	0.213	0.169	*0.121*	**0.076**
c020	*0.123*	**0.063**	0.39	0.185
c021	0.046	*0.043*	0.471	**0.007**
c022	**0.016**	*0.036*	0.402	0.063
c023	0.209	*0.119*	**0.053**	0.14
c024	0.189	**0.052**	*0.138*	0.176
c025	0.215	**0.097**	0.206	*0.117*
c026	0.325	*0.09*	**0.087**	0.263
c027	0.122	**0.017**	0.179	*0.086*
c028	0.317	**0.046**	0.274	*0.152*
c029	0.204	*0.099*	**0.071**	0.192
c030	*0.07*	**0.051**	0.567	0.205
c031	0.154	**0.058**	*0.1*	0.17
c032	0.251	**0.042**	*0.072*	0.111
c033	0.277	0.152	*0.135*	**0.078**
c034	*0.065*	0.114	0.691	**0.011**
c035	0.327	*0.212*	**0.011**	0.357
c036	0.175	0.115	*0.067*	**0.045**
c037	**0.036**	*0.041*	0.631	0.21
c038	**0.035**	*0.041*	0.847	0.262
c039	*0.176*	**0.064**	0.307	0.198
c040	0.15	**0.055**	0.146	*0.119*

Task	Sample+Weight ($s = 5$)	Full model ($s = 5$)	HL ($s = 5$)	HL ($s = 500K$)
c041	**0.041**	*0.044*	0.446	0.071
c042	**0.008**	0.076	0.453	*0.063*
c043	*0.003*	**0.001**	0.605	0.042
c044	*0.015*	**0.014**	0.463	0.033
c045	*0.003*	0.022	*0.003*	**0.002**
c046	0.05	**0.026**	0.381	*0.048*
c047	*0.086*	**0.06**	0.36	0.091
c048	0.015	**0.009**	0.778	*0.011*
c049	**0.019**	*0.031*	0.067	0.073
c050	*0.018*	0.029	0.651	**0.012**
c051	*0.064*	**0.059**	0.298	0.077
c052	0.089	**0.011**	0.357	*0.012*
c053	0.295	*0.052*	0.371	**0.041**
c054	0.237	**0.018**	*0.095*	0.212
c055	0.32	*0.088*	0.302	**0.062**
c056	0.127	**0.044**	0.14	*0.118*
c057	0.238	**0.056**	0.112	*0.097*
c058	**0.06**	0.081	*0.063*	0.242
c059	0.475	0.247	**0.037**	*0.197*
c060	0.197	0.136	**0.064**	*0.1*
c061	**0.002**	*0.143*	0.847	0.155
c062	*0.087*	**0.065**	0.554	0.141
c063	0.081	**0.03**	*0.039*	0.087
c064	*0.329*	**0.062**	0.536	0.536
c065	*0.181*	**0.113**	0.372	0.339
c066	*0.106*	**0.085**	0.453	0.453
c067	*0.077*	**0.037**	0.491	0.491
c068	0.095	*0.067*	0.774	**0.037**
c069	*0.132*	**0.105**	0.392	0.258
c070	*0.044*	**0.036**	0.668	0.668
c071	*0.068*	**0.06**	0.596	0.255
c072	**0.007**	*0.011*	0.776	0.099
c073	0.318	0.294	**0.174**	**0.174**
c074	0.33	**0.098**	*0.115*	0.116

Task	Sample+Weight ($s = 5$)	Full model ($s = 5$)	HL ($s = 5$)	HL ($s = 500K$)
c075	0.077	**0.027**	0.373	*0.028*
c076	0.286	0.124	**0.019**	**0.019**
c077	0.284	*0.232*	0.451	**0.202**
c078	*0.134*	**0.061**	0.409	0.319
c079	0.712	**0.319**	0.817	*0.419*
c080	**0.003**	*0.014*	0.887	0.887
c081	0.131	*0.116*	0.548	**0.109**
c082	*0.139*	**0.104**	0.229	0.188
c083	0.256	*0.122*	0.2	**0.064**
c084	0.081	0.058	*0.052*	**0.031**
c085	0.259	**0.063**	0.205	*0.196*
c086	0.303	**0.046**	0.193	*0.127*
c087	0.149	*0.086*	0.23	**0.056**
c088	0.218	**0.092**	*0.155*	0.19
c089	0.141	0.184	*0.047*	**0.011**
c090	*0.028*	**0.017**	0.304	0.056
c091	*0.037*	0.047	0.29	**0.022**
c092	*0.176*	*0.176*	0.187	**0.103**
c093	0.039	*0.037*	0.54	**0.011**
c094	0.243	0.173	*0.145*	**0.116**
c095	*0.396*	**0.058**	0.59	0.604
c096	*0.061*	**0.044**	0.537	0.416
c097	*0.132*	**0.092**	0.598	0.486
c098	**0.108**	*0.111*	0.508	0.508
c099	*0.189*	**0.056**	0.336	0.336
c100	*0.026*	**0.02**	0.744	0.695

A.3 Choice of representative subset of tasks

For development of the modeling work, we constructed a difficult subset of the 100 original List Function tasks. Due to high API costs of LLMs, for **(TODO: cite)**, we limited analysis to this subset in order to limit the expenses of the project. Our 20-task subset consists of 5 tasks sampled from the "representative subset" in **(TODO: cite appendix of JR thesis)** (c045, c090, c025, c029, c033), the 3 hardest tasks based on average human accuracy (c015,

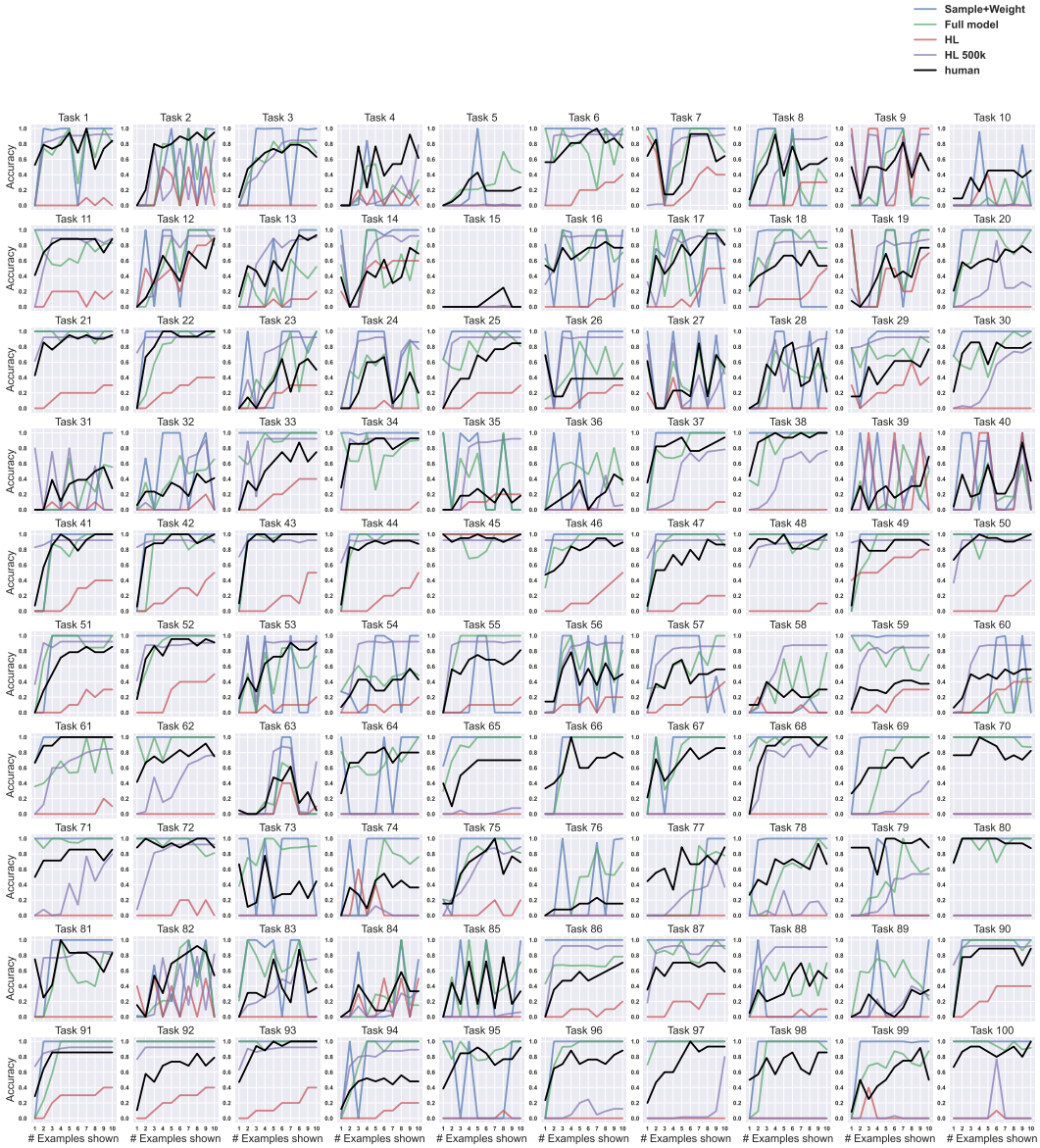

Figure A.1: Model accuracy and human accuracy for all 100 tasks, by example.

c076, c035), the 3 tasks with the highest variance in mean human accuracy between examples (c085, c027, c004), the 3 tasks where HL 500k was furthest from mean human accuracy (c080, c100, c070), the 3 tasks where GPT-4 with 5 samples was the furthest from mean human accuracy (c002, c079, c061), and the 3 tasks where GPT-3.5 with 10 samples was the furthest from mean human accuracy (c002, c079, c060).

Appendix B

Bootstrapping a DSL Supplement

B.1 Prompts

For these prompts, there was very little "prompt engineering", and we suspect further optimization (whether automatic or by hand) could yield further performance improvements.

B.1.1 Solving individual tasks with a LLM

```
Write a Python function `fn` that implements the underlying rule
   mapping each input to its output.

Here are examples of the transformation:
{input-output examples}

Here are examples of other transformations and the code for the
   underlying rule:
{few shot examples}

Now, determine the underlying rule for the earlier examples and
   write a Python function `fn` that implements this rule.

Do not import  any libraries.
```

B.1.2 Translating Python to DSL primitives

```
# Domain description
```

```
{domain_description}

# Problem description

Here are examples of the transformation:
{examples}

Here is the python code that solves the task.
```
{python_code}
```

# DSL description

These are the types in your DSL:
{existing_types}

# Task description
Define all the new primitives (functions and/or constants) that you will need in order to translate the task specified above. Each primitive declaration should have the following format: `Primitive(<primitive name>, <primitive type>, <curried primitive_function>)`. Then, write a program in De-Brujin indexed lambda calculus using these primitives.

# Final instructions
Remember, *anything* you use in the solution will have to be defined, even basic control flow functions (like "if") and constants (like "s").

Do not put quotes around primitives in the final program. Follow the exact format of previous examples. The program should be a purely function De-Brujin indexed lambda calculus (do not use `let`).
```

B.2 Induced Libraries

Here, we list the entirety of the libraries induced by the model. For each domain, we list the primitive procedures, then the primitive constants, then the primitives learned via compression. We show the primitive name (e.g. `concat`), then the primitive declaration, which shows the name (e.g. `"concat"`), type (e.g. `arrow(tlist(t0), tlist(t0), tlist(t0))`), and definition (e.g. `lambda x: lambda y: lambda x + y`).

Note that, in the type definitions, `t0` and `t1` represent type variables and `arrow` represents a function. So, the type `arrow(tlist(t0), tlist(t0), tlist(t0))` represents a function that takes in two arguments, both generic list-types, and returns a list of the same type. The function definitions are curried, so the function `add(x, y)` is represented as `add(x)(y)`.

For some of the compressed primitives (starting with `#`), we also include a note for what it evaluates to to improve readability. Note that this can be done (as in LILO [32]) automatically.

B.2.1 List Functions

```
Primitive procedures:

-
Primitive("-", arrow(tint, tint, tint), lambda x: lambda y: x - y)

<
Primitive("<", arrow(tint, tint, tbool), lambda x: lambda y: x < y)

>=
Primitive(">=", arrow(tint, tint, tbool), lambda x: lambda y: x >= y)

concat
Primitive("concat", arrow(tlist(t0), tlist(t0), tlist(t0)), lambda x:
    lambda y: x + y)

filter
Primitive("filter", arrow(arrow(tint, tbool), tlist(tint), tlist(tint))
    , lambda f: lambda lst: [x for x in lst if f(x)])

get_item
```

```
Primitive("get_item", arrow(tlist(tint), tint, tint), lambda l: lambda
    i: l[i])

if
Primitive("if", arrow(tbool, t0, t0, t0), lambda c: lambda x: lambda y:
     x if c else y)

insert
Primitive("insert", arrow(tlist(tint), tint, tint, tlist(tint)), lambda
     lst: lambda idx: lambda val: lst[:idx] + [val] + lst[idx:])

int->list
Primitive("int->list", arrow(tint, tlist(tint)), lambda x: [x])

is_even_index
Primitive("is_even_index", arrow(tint, tbool), lambda i: i % 2 == 0)

length
Primitive("length", arrow(tlist(tint), tint), lambda l: len(l))

list
Primitive("list", arrow(t0, arrow(t0, arrow(t0, tlist(t0)))), lambda x:
     lambda y: lambda z: [x, y, z])

map_acc
Primitive("map_acc", arrow(tint, arrow(tint, tint, tint), tlist(tint),
    tlist(tint)), lambda acc: lambda f: lambda lst: [f(acc + i)(x) for i
    , x in enumerate(lst)])

map
Primitive("map", arrow(arrow(t0, tlist(t0)), tlist(t0), tlist(t0)),
    lambda f: lambda lst: [item for x in lst for item in f(x)])

max
Primitive("max", arrow(tlist(tint), tint), lambda lst: max(lst))

replicate
Primitive("replicate", arrow(tint, tint, tlist(tint)), lambda n: lambda
```

```
    x: [x] * n)

reverse
Primitive("reverse", arrow(tlist(t0), tlist(t0)), lambda lst: lst[::
    -1])

set_item
Primitive("set_item", arrow(tlist(tint), tint, tint, tlist(tint)),
    lambda l: lambda i: lambda v: l[:i] + [v] + l[i+1:])

sum
Primitive("sum", arrow(tlist(tint), tint), lambda lst: sum(lst))

swap
Primitive("swap", arrow(tlist(tint), tlist(tint)), lambda lst: lst[:1]
    + [lst[2], lst[1]] + lst[3:] if len(lst) >= 3 else lst)

tail
Primitive("tail", arrow(tlist(tint), tlist(tint)), lambda l: l[1:])

Primitive constants:

37
Primitive("37", tint, 37)

77
Primitive("77", tint, 77)

const_list_1943258049
Primitive("const_list_1943258049", tlist(tint), [1, 9, 4, 3, 2, 5, 8,
    0, 4, 9])

const_list_81_99_41_23_22_75_68_30_24_69
Primitive("const_list_81_99_41_23_22_75_68_30_24_69", tlist(tint), [81,
     99, 41, 23, 22, 75, 68, 30, 24, 69])

fixed_list
Primitive("fixed_list", tlist(tint), [9, 6, 3, 8, 5])
```

```

list11_21_43_19
Primitive("list11_21_43_19", tlist(tint), [11, 21, 43, 19])

list7_89_0_57
Primitive("list7_89_0_57", tlist(tint), [7, 89, 0, 57])

list_7291
Primitive("list_7291", tlist(tint), [7, 2, 9, 1])

list_9340
Primitive("list_9340", tlist(tint), [9, 3, 4, 0])

rule
Primitive("rule", tlist(tint), [7, 3, 8, 4, 3])

rule_list
Primitive("rule_list", tlist(tint), [92, 63, 34, 18, 55])

Compressed primitives

#(- #(length (tail list_7291)) (length (int->list #(length (tail
   list_7291)))))
Note: Evaluates to the integer 3.

#(lambda (filter (lambda (< $0 #(- #(length (tail list_7291)) (length (
   int->list #(length (tail list_7291)))))))) $0))
Note: Gets all elements in the input with value less than 3.

#(lambda (filter (lambda (< (- $0 #(length (tail list_7291))) #(length
   (tail list_7291)))) $0))
Note: Gets all elements in the input with value less than 5.

#(lambda (get_item $0 (- 77 77)))
Note: Gets the first element of the input.

#(lambda (int->list (#(lambda (get_item $0 (- 77 77))) $0)))
Note: Evaluates to the list containing the first element of the input.
```

```

#(lambda (int->list (get_item (filter (lambda (is_even_index $0)) $0)
   #(length (tail list_7291)))))
Note: Evaluates to the list containing the tenth element of the input.

#(lambda (lambda (filter (lambda (< $0 $1)) (tail $1))))
Note: Gets all the elements in the second list that are bigger than the
    given integer, skipping the first element.

#(lambda (lambda (lambda (concat (int->list $0) (concat (int->list $1)
   (concat (int->list $2) (replicate (- 77 77) #(length (tail list_7291
   ))))))))))
Note: Makes a list out of three integers, padded with four zeros.

#(length (int->list 77))
Note: Evaluates to the integer 1.

#(length (tail list_7291))
Note: Evaluates to the integer 4.
```

B.2.2 Regular Expressions

```
Primitive procedures:

concat
Primitive("concat", arrow(tstr, tstr, tstr), lambda x: lambda y: x + y)

if
Primitive("if", arrow(tbool, t0, t0, t0), lambda c: lambda x: lambda y:
    x if c else y)

in
Primitive("in", arrow(tstr, tstr, tbool), lambda x: lambda y: x in y)

length
Primitive("length", arrow(tstr, tint), lambda s: len(s))

map
```

```
Primitive("map", arrow(tstr, arrow(tstr, tstr), tstr), lambda s: lambda
    f: ''.join(f(c) for c in s))

not_in
Primitive("not_in", arrow(tstr, tstr, tbool), lambda x: lambda y: x not
    in y)

repeat
Primitive("repeat", arrow(tstr, tint, tstr), lambda s: lambda n: s * n)

replace
Primitive("replace", arrow(tstr, tstr, tstr, tstr), lambda s: lambda
   old: lambda new: s.replace(old, new))

slice
Primitive("slice", arrow(tstr, tint, tint, tstr), lambda s: lambda
   start: lambda end: s[start:end])

subtract
Primitive("subtract", arrow(tint, tint, tint), lambda x: lambda y: x -
   y)

Primitive constants:

char_b
Primitive("char_b", tchar, 'b')

cv
Primitive("cv", tstr, "cv")

d
Primitive("d", tstr, "d")

ee
Primitive("ee", tstr, "ee")

f
Primitive("f", tstr, "f")
```

```
49
50 g
51 Primitive("g", tstr, "g")
52
53 j
54 Primitive("j", tstr, "j")
55
56 k
57 Primitive("k", tstr, "k")
58
59 lq
60 Primitive("lq", tstr, "lq")
61
62 na
63 Primitive("na", tstr, "na")
64
65 ou
66 Primitive("ou", tstr, "ou")
67
68 p
69 Primitive("p", tstr, "p")
70
71 ql
72 Primitive("ql", tstr, "ql")
73
74 r
75 Primitive("r", tstr, "r")
76
77 tm
78 Primitive("tm", tstr, "tm")
79
80 vowels
81 Primitive("vowels", tstr, "aeiou")
82
83 wi
84 Primitive("wi", tstr, "wi")
85
86 x
```

```
Primitive("x", tstr, "x")

Compress primitives:

#(lambda (#(lambda (lambda (lambda (slice $0 (length $1) (length $2)))) ) cv d $0))
Note: Gets the second element of the string.

#(lambda (lambda (#(lambda (lambda (lambda (concat (slice $0 #(subtract (length d) (length d)) (subtract #(subtract (length d) (length d)) (length $1))) $2)))) $0 d $1)))
Note: Takes 2 arguments x and y. Gets the first (len(x)-1) elements of x, and concatenates y to it.

#(lambda (lambda (#(lambda (lambda (lambda (concat (slice $0 0 ( subtract 0 (length $1))) $2)))) $0 d $1)))

#(lambda (lambda (concat $0 (#(lambda (slice $0 (length d) (length $0)) ) $1))))
Note: Concats the first letter of a string and a different string.

#(lambda (lambda (if (in (slice $1 (length d) (length cv)) vowels) ( concat (slice $0 #(subtract (length d) (length d)) (length d)) $1) $1)))
Note: Takes 2 arguments x and y. If the second element of x is a vowel, concatenate the first letter of x to y. Otherwise, return y.

#(lambda (lambda (lambda (concat (slice $0 #(subtract (length d) ( length d)) (subtract #(subtract (length d) (length d)) (length $1))) $2))))
Note: Takes 3 arguments x, y, and z. Gets the first (len(x)-len(y)) elements of x, and concatenates z to it.

#(lambda (lambda (lambda (if (in $0 vowels) (concat $1 $2) $2))))
Note: Concatenates string 1 with string2 if the given character is a vowel, otherwise returns string2.

#(lambda (lambda (lambda (slice $0 (length $1) (length $2)))))
```

```
Note: Gets the slice of the string between the length of list 1 and the
    length of list 2.

#(lambda (slice $0 #(subtract (length d) (length d)) (length d)))
Note: Identity function.

#(lambda (slice $0 (length d) (length $0)))
Note: Gets the tail of a string (e.g. string[1:]).

#(subtract (length d) (length d))
Note: Evaluates to the integer 0.
```

B.2.3 CLEVR

```
Primitive procedures:

==
Primitive("==", arrow(t0, t0, tbool), lambda x: lambda y: x == y)

>
Primitive(">", arrow(tint, tint, tbool), lambda x: lambda y: x > y)

any
Primitive("any", arrow(tlist(t0), arrow(t0, tbool), tbool), lambda lst:
    lambda pred: any(pred(x) for x in lst))

filter
Primitive("filter", arrow(tlist(t0), arrow(t0, tbool), tlist(t0)),
   lambda lst: lambda cond: [x for x in lst if cond(x)])

get
Primitive("get", arrow(tclevrobject, tstring, t0), lambda obj: lambda
   key: obj[key])

get_color
Primitive("get_color", arrow(tclevrobject, tclevrcolor), lambda obj:
   obj['color'])

```

```
get_shape
Primitive("get_shape", arrow(tclevrobject, tclevrshape), lambda obj:
    obj['shape'])

if
Primitive("if", arrow(tbool, t0, t0, t0), lambda c: lambda x: lambda y:
     x if c else y)

length
Primitive("length", arrow(tlist(t0), tint), lambda x: len(x))

map
Primitive("map", arrow(arrow(tclevrobject, t0), tlist(tclevrobject),
    tlist(t0)), lambda f: lambda lst: [f(x) for x in lst])

Primitive constants:

brown
Primitive("brown", tclevrcolor, "brown")

cube
Primitive("cube", tclevrshape, "cube")

gray_green_brown_brown_green_cyan_gray_red_gray_gray
Primitive("gray_green_brown_brown_green_cyan_gray_red_gray_gray", tlist
    (tclevrcolor), ['gray', 'green', 'brown', 'brown', 'green', 'cyan',
    'gray', 'red', 'gray', 'gray'])

gray_red_blue_brown
Primitive("gray_red_blue_brown", tlist(tclevrcolor), ['gray', 'red', '
    blue', 'brown'])

green
Primitive("green", tclevrcolor, "green")

large
Primitive("large", tclevrsize, "large")

```

```
material
Primitive("material", tstring, "material")

metal
Primitive("metal", tclevrmaterial, "metal")

red
Primitive("red", tclevrcolor, "red")

rubber
Primitive("rubber", tclevrmaterial, "rubber")

size
Primitive("size", tstring, "size")

small
Primitive("small", tclevrsize, "small")

sphere
Primitive("sphere", tclevrshape, "sphere")

Compressed primitives:

#(lambda (== (filter
   gray_green_brown_brown_green_cyan_gray_red_gray_gray (lambda (#(
   lambda (lambda (any (filter $0 (lambda (== $2 (get_color $0)))) (
   lambda (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2)))))
   small small (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2)))
   )) sphere (get_shape $0) (== metal metal))))))) $0 $1))) (filter
   gray_red_blue_brown (lambda (> #(length (filter gray_red_blue_brown
   (lambda (== $0 red)))) #(length (filter gray_red_blue_brown (lambda
   (== $0 red)))))))))

#(lambda (if (any $0 (lambda (#(lambda (lambda (lambda (if (== $1 $2)
   $0 (== $1 $2))))) (get_color $0) (get_color $0) (#(lambda (lambda (
   lambda (if (== $1 $2) $0 (== $1 $2))))) large (get $0 size) (#(
   lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2))))) cube (
   get_shape $0) (== (get $0 material) metal)))))) large small))
```

```
79
80 #(lambda (lambda (any (filter $0 (lambda (== $2 (get_color $0)))) (
   lambda (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2)))))
   small small (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2)))
   )) sphere (get_shape $0) (== metal metal)))))))
81
82 #(lambda (lambda (lambda (#(lambda (lambda (lambda (if (== $1 $2) $0
   (== $1 $2))))) small (get $1 size) (#(lambda (lambda (lambda (if (==
    $1 $2) $0 (== $1 $2))))) $0 (get_shape $1) (== $2 metal))))))
83
84 #(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2)))))
85 Note: Takes arguments x, y, and z. If x == y, returns z. Otherwise
   returns False.
86
87 #(lambda (lambda (lambda (if (any $0 (lambda (#(lambda (lambda (lambda
   (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2))))) small (
   get $1 size) (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2))
   ))) $0 (get_shape $1) (== $2 metal)))))) (get $0 material) $0 (
   get_shape $0)))) $1 $2))))
88
89 #(lambda (lambda (lambda (if (any $0 (lambda (#(lambda (lambda (lambda
   (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2))))) small (
   get $1 size) (#(lambda (lambda (lambda (if (== $1 $2) $0 (== $1 $2))
   ))) $0 (get_shape $1) (== $2 metal)))))) metal $0 cube))) $1 $2))))
90
91 #(lambda (lambda (lambda (length (filter $0 (lambda (== (get $0 $2) $3)
   ))))))
92
93 #(lambda (lambda (length (filter $0 (lambda (#(lambda (lambda (lambda (
   if (== $1 $2) $0 (== $1 $2))))) (get_color $0) red (#(lambda (lambda
    (lambda (if (== $1 $2) $0 (== $1 $2))))) $2 (get_shape $0) (== (get
    $0 material) metal)))))))
94
95 #(length (filter gray_red_blue_brown (lambda (== $0 red))))
```

References

[1] W. James, *The Principles of Psychology* (Dover Books v. 1-2). Dover Publications, 1950, ISBN: 9780486203829. URL: https://books.google.com/books?id=5q1kDQAAQBAJ.

[2] J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. Thagard, *Induction: Processes Of Inference*. The MIT Press, Mar. 1989, ISBN: 9780262275576. DOI: 10.7551/mitpress/3729.001.0001. URL: https://doi.org/10.7551/mitpress/3729.001.0001.

[3] A. F. Chalmers, *What is This Thing Called Science?: An Assessment of the Nature and Status of Science and its Methods*. Indianapolis: Univ. Of Queensland Press, 1976.

[4] D. Hume, "An enquiry concerning human understanding.," in (Essays and treatises on several subjects, Vol 2: Containing An enquiry concerning human understanding, A dissertation on the passions, An enquiry concerning the principles of morals, and The natural history of religion.), Essays and treatises on several subjects, Vol 2: Containing An enquiry concerning human understanding, A dissertation on the passions, An enquiry concerning the principles of morals, and The natural history of religion. Unknown Publisher, 1779, pp. 3–212. DOI: 10.1037/11713-001. URL: https://doi.org/10.1037/11713-001.

[5] T. L. Griffiths and J. B. Tenenbaum, "Optimal predictions in everyday cognition," en, *Psychol Sci*, vol. 17, no. 9, pp. 767–773, Sep. 2006.

[6] F. Lieder and T. L. Griffiths, "Resource-rational analysis: Understanding human cognition as the optimal use of limited computational resources," en, *Behav Brain Sci*, vol. 43, e1, Feb. 2019.

[7] C. G. Correa, M. K. Ho, F. Callaway, and T. L. Griffiths, *Resource-rational task decomposition to minimize planning costs*, 2020. arXiv: 2007.13862 [cs.AI].

[8] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, *A survey on in-context learning*, 2023. arXiv: 2301.00234 [cs.CL].

[9] D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. The MIT Press, Jul. 2010, ISBN: 9780262514620. DOI: 10.7551/mitpress/9780262514620.001.0001. URL: https://doi.org/10.7551/mitpress/9780262514620.001.0001.

[10] L. Wong, G. Grand, A. K. Lew, N. D. Goodman, V. K. Mansinghka, J. Andreas, and J. B. Tenenbaum, *From word models to world models: Translating from natural language to the probabilistic language of thought*, 2023. arXiv: 2306.12672 [cs.CL].

[11] K. Ellis, C. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum, *Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning*, 2020. arXiv: 2006.08381 [cs.AI].

[12] J. S. Rule, S. T. Piantadosi, and J. B. Tenenbaum, "Efficient learning of symbolic concepts via metaprogram search,"

[13] I. Dasgupta, E. Schulz, and S. J. Gershman, "Where do hypotheses come from?" *Cognitive Psychology*, vol. 96, pp. 1–25, 2017, ISSN: 0010-0285. DOI: https://doi.org/10.1016/j.cogpsych.2017.05.001. URL: https://www.sciencedirect.com/science/article/pii/S0010028516302766.

[14] J.-P. Fränken, N. C. Theodoropoulos, and N. R. Bramley, "Algorithms of adaptation in inductive inference," *Cognitive Psychology*, vol. 137, p. 101 506, 2022, ISSN: 0010-0285. DOI: https://doi.org/10.1016/j.cogpsych.2022.101506. URL: https://www.sciencedirect.com/science/article/pii/S0010028522000421.

[15] K. Ellis, *Human-like few-shot learning via bayesian reasoning over natural language*, 2023. arXiv: 2306.02797 [cs.CL].

[16] W. T. Piriyakulkij and K. Ellis, *Doing experiments and revising rules with natural language and probabilistic reasoning*, 2024. arXiv: 2402.06025 [cs.AI].

[17] R. M. Haefner, P. Berkes, and J. Fiser, "Perceptual decision-making as probabilistic inference by neural sampling," *Neuron*, vol. 90, no. 3, pp. 649–660, May 2016, ISSN: 0896-6273. DOI: 10.1016/j.neuron.2016.03.020. URL: https://doi.org/10.1016/j.neuron.2016.03.020.

[18] J. Fiser, P. Berkes, G. Orbán, and M. Lengyel, "Statistically optimal perception and learning: From behavior to neural representations," *Trends in Cognitive Sciences*, vol. 14, no. 3, pp. 119–130, Mar. 2010, ISSN: 1364-6613. DOI: 10.1016/j.tics.2010.01.003. URL: https://doi.org/10.1016/j.tics.2010.01.003.

[19] J. Austin, A. Odena, M. Nye, *et al.*, *Program synthesis with large language models*, 2021. arXiv: 2108.07732 [cs.PL].

[20] Y. Li, D. Choi, J. Chung, *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, ISSN: 1095-9203. DOI: 10.1126/science.abq1158. URL: http://dx.doi.org/10.1126/science.abq1158.

[21] OpenAI, J. Achiam, S. Adler, *et al.*, *Gpt-4 technical report*, 2024. arXiv: 2303.08774 [cs.CL].

[22] Y. Li, J. Parsert, and E. Polgreen, *Guiding enumerative program synthesis with large language models*, 2024. arXiv: 2403.03997 [cs.AI].

[23] N. Butt, B. Manczak, A. Wiggers, C. Rainone, D. Zhang, M. Defferrard, and T. Cohen, *Codeit: Self-improving language models with prioritized hindsight replay*, 2024. arXiv: 2402.04858 [cs.AI].

[24] D. Brandfonbrener, S. Raja, T. Prasad, C. Loughridge, J. Yang, S. Henniger, W. E. Byrd, R. Zinkov, and N. Amin, *Verified multi-step synthesis using large language models and monte carlo tree search*, 2024. arXiv: 2402.08147 [cs.SE].

[25] T. Sechopoulos, "Program synthesis with symbolic properties," M.S. thesis, Massachusetts Institute of Technology, 2022.

[26] A. Odena and C. Sutton, *Learning to represent programs with property signatures*, 2020. arXiv: 2002.09030 [cs.PL].

[27] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," *SIGPLAN Not.*, vol. 50, no. 6, pp. 229–239, Jun. 2015, ISSN: 0362-1340. DOI: 10.1145/2813885.2737977. URL: https://doi.org/10.1145/2813885.2737977.

[28] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, *Voyager: An open-ended embodied agent with large language models*, 2023. arXiv: 2305.16291 [cs.AI].

[29] Z. Wang, D. Fried, and G. Neubig, *Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks*, 2024. arXiv: 2401.12869 [cs.AI].

[30] E. Stengel-Eskin, A. Prasad, and M. Bansal, *Regal: Refactoring programs to discover generalizable abstractions*, 2024. arXiv: 2401.16467 [cs.SE].

[31] C. Wong, K. Ellis, J. B. Tenenbaum, and J. Andreas, *Leveraging language to learn program abstractions and search heuristics*, 2022. arXiv: 2106.11053 [cs.LG].

[32] G. Grand, L. Wong, M. Bowers, T. X. Olausson, M. Liu, J. B. Tenenbaum, and J. Andreas, *Lilo: Learning interpretable libraries by compressing and documenting code*, 2024. arXiv: 2310.19791 [cs.CL].

[33] M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, and A. Solar-Lezama, "Top-down synthesis for library learning," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1182–1213, Jan. 2023, ISSN: 2475-1421. DOI: 10.1145/3571234. URL: http://dx.doi.org/10.1145/3571234.

[34] N. R. Bramley and F. Xu, "Active inductive inference in children and adults: A constructivist perspective," *Cognition*, vol. 238, p. 105 471, 2023, ISSN: 0010-0277. DOI: https://doi.org/10.1016/j.cognition.2023.105471. URL: https://www.sciencedirect.com/science/article/pii/S0010027723001051.

[35] J.-Q. Zhu, J. Sundh, J. Spicer, N. Chater, and A. N. Sanborn, "The autocorrelated bayesian sampler: A rational process for probability judgments, estimates, confidence intervals, choices, confidence judgments, and response times.," *Psychological Review*, vol. 131, no. 2, pp. 456–493, 2024. DOI: 10.1037/rev0000427. URL: https://doi.org/10.1037/rev0000427.

[36] J. B. Tenenbaum, "A bayesian framework for concept learning," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[37] T. Webb, K. J. Holyoak, and H. Lu, *Emergent analogical reasoning in large language models*, 2023. arXiv: 2212.09196 [cs.AI].

[38] T. Webb, K. J. Holyoak, and H. Lu, *Evidence from counterfactual tasks supports emergent analogical reasoning in large language models*, 2024. arXiv: 2404.13070 [cs.CL].

[39] G. Gendron, Q. Bao, M. Witbrock, and G. Dobbie, *Large language models are not strong abstract reasoners*, 2024. arXiv: 2305.19555 [cs.CL].

[40] M. Lewis and M. Mitchell, *Using counterfactual tasks to evaluate the generality of analogical reasoning in large language models*, 2024. arXiv: 2402.08955 [cs.AI].

[41] A. Moskvichev, V. V. Odouard, and M. Mitchell, *The conceptarc benchmark: Evaluating understanding and generalization in the arc domain*, 2023. arXiv: 2305.07141 [cs.LG].

[42] L. Qiu, L. Jiang, X. Lu, *et al.*, *Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement*, 2024. arXiv: 2310.08559 [cs.CL].

[43] R. Wang, E. Zelikman, G. Poesia, Y. Pu, N. Haber, and N. D. Goodman, *Hypothesis search: Inductive reasoning with language models*, 2023. arXiv: 2309.05660 [cs.LG].

[44] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, *Is self-repair a silver bullet for code generation?* 2024. arXiv: 2306.09896 [cs.CL].

[45] A. N. Sanborn, T. L. Griffiths, and D. J. Navarro, "Rational approximations to rational models: Alternative algorithms for category learning," en, *Psychol. Rev.*, vol. 117, no. 4, pp. 1144–1167, Oct. 2010.

[46] Y. Yang and S. T. Piantadosi, "One model for the learning of language," *Proceedings of the National Academy of Sciences*, vol. 119, no. 5, e2021865119, 2022. DOI: 10.1073/pnas.2021865119. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2021865119. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2021865119.

[47] A. Cropper and S. H. Muggleton, *Metagol system*, https://github.com/metagol/metagol, 2016. URL: https://github.com/metagol/metagol.

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, *Chain-of-thought prompting elicits reasoning in large language models*, 2023. arXiv: 2201.11903 [cs.CL].

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

[50] Q. Zhang, C. Singh, L. Liu, X. Liu, B. Yu, J. Gao, and T. Zhao, *Tell your model where to attend: Post-hoc attention steering for llms*, 2023. arXiv: 2311.02262 [cs.CL].

[51] H. Touvron, L. Martin, K. Stone, *et al.*, *Llama 2: Open foundation and fine-tuned chat models*, 2023. arXiv: 2307.09288 [cs.CL].

[52] S. J. Han, K. Ransom, A. Perfors, and C. Kemp, *Inductive reasoning in humans and large language models*, 2023. arXiv: 2306.06548 [cs.CL].

[53] J. Andreas, D. Klein, and S. Levine, "Learning with latent language," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds., New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2166–2179. DOI: 10.18653/v1/N18-1197. URL: https://aclanthology.org/N18-1197.

[54] J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick, "Clevr: A diagnostic dataset for compositional language and elementary visual reasoning," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1988–1997. DOI: 10.1109/CVPR.2017.215.