A Appendix

A.1 Supplemental Results

Fig. 6 illustrates model predictions across every Number Game concept in [33].

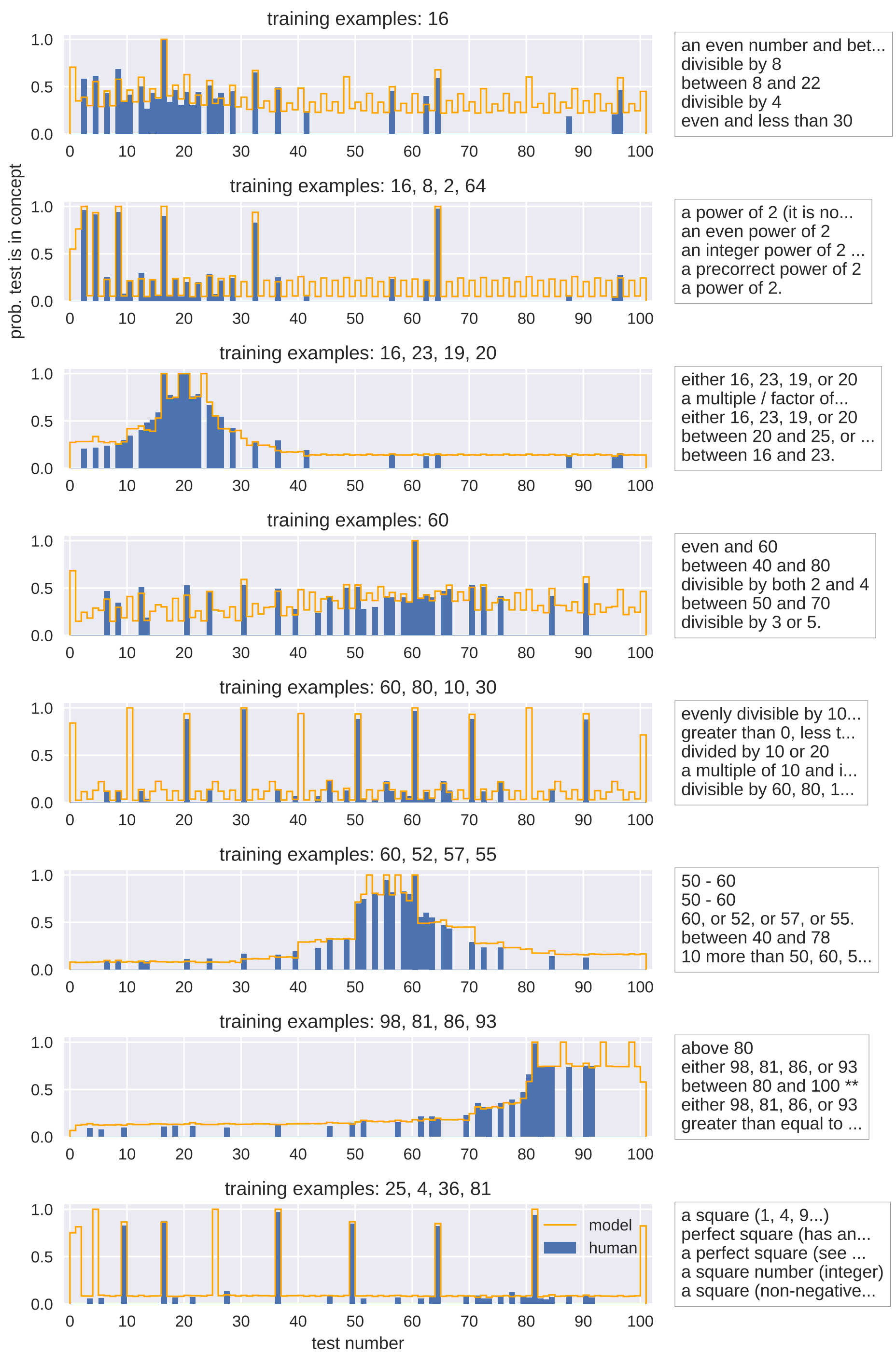

Figure 6: Model predictions across every Number Game concept in [33]

Recall that we deduplicated the proposals instead of performing actual importance sampling. Fig. 7 contrasts model fit for importance sampling and deduplication. We originally did deduplication simply because importance sampling is not possible with GPT-4, and GPT-4 proved necessary for the logical concepts. On number concepts we used code-davinci-002, from which we can construct an importance sampler because it exposes the log probability of its samples. On number concepts deduplication provides a fit that is on-par (actually slightly better) compared to importance sampling (Fig. 7).

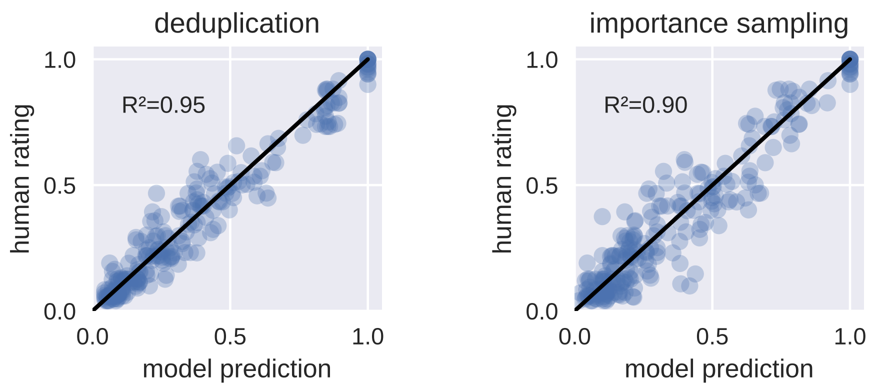

Figure 7: Monte Carlo inference using deduplication instead of importance sampling does not harm model fit to human data. The above figures show Number Game models using a learned prior and 100 samples, and show predictions only on holdout data.

A.2 Human Study

16 participants were recruited primarily through a Slack message sent to a channel populated by members of our academic department. Participants had an average age 28.1 (stddev 13.6, all over 18), and were 7 male/5 female/1 nonbinary/3 declined to answer. Participants were randomly split between the concepts of *most common color* / *least common color*. Each participant went through 15 trials, and took an average of 294s to complete those 15 trials. In exchange for participating in the study, participants received $10 in Amazon gift cards. Fig. 8 illustrates the web interface shown to our human participants, including the cover story.

A.3 Modeling

A.3.1 Temperature and Platt Transform

Adding a temperature parameter T to a model corresponds to computing the posterior via

$$p_{\text{Temp}}(X_{\text{test}} \in C|X_{1:K}) \approx \sum_{C \in \{C^{(1)},\ldots,C^{(S)}\}} w^{(C)} \mathbb{1}\left[X_{\text{test}} \in C\right], \text{ where}$$

$$w^{(C)} = \frac{\left(\tilde{w}^{(C)}\right)^{1/T}}{\sum_{C'} \left(\tilde{w}^{(C')}\right)^{1/T}} \text{ and } \tilde{w}^{(C)} = p(C)p(X_{1:K}|C)\mathbb{1}\left[C \in \{C^{(1)},\ldots,C^{(S)}\}\right] \tag{10}$$

Adjusting the predictions of a model using a Platt transform corresponds to introducing parameters a and b which transform the predictions according to

$$p_{\text{Platt}}(X_{\text{test}} \in C|X_{1:K}) = \text{Logistic}(b + a \times \text{Logistic}^{-1}\left(p(X_{\text{test}} \in C|X_{1:K})\right)) \tag{11}$$

For the number game, every model has its outputs transformed by a learned Platt transform. This is because we are modeling human ratings instead of human responses. We expect that the ratings correspond to some monotonic transformation of the human's subjective probability estimates, and so this transformation gives some extra flexibility by inferring the correspondence between probabilities and ratings. Logical concept models do not use Platt transforms.

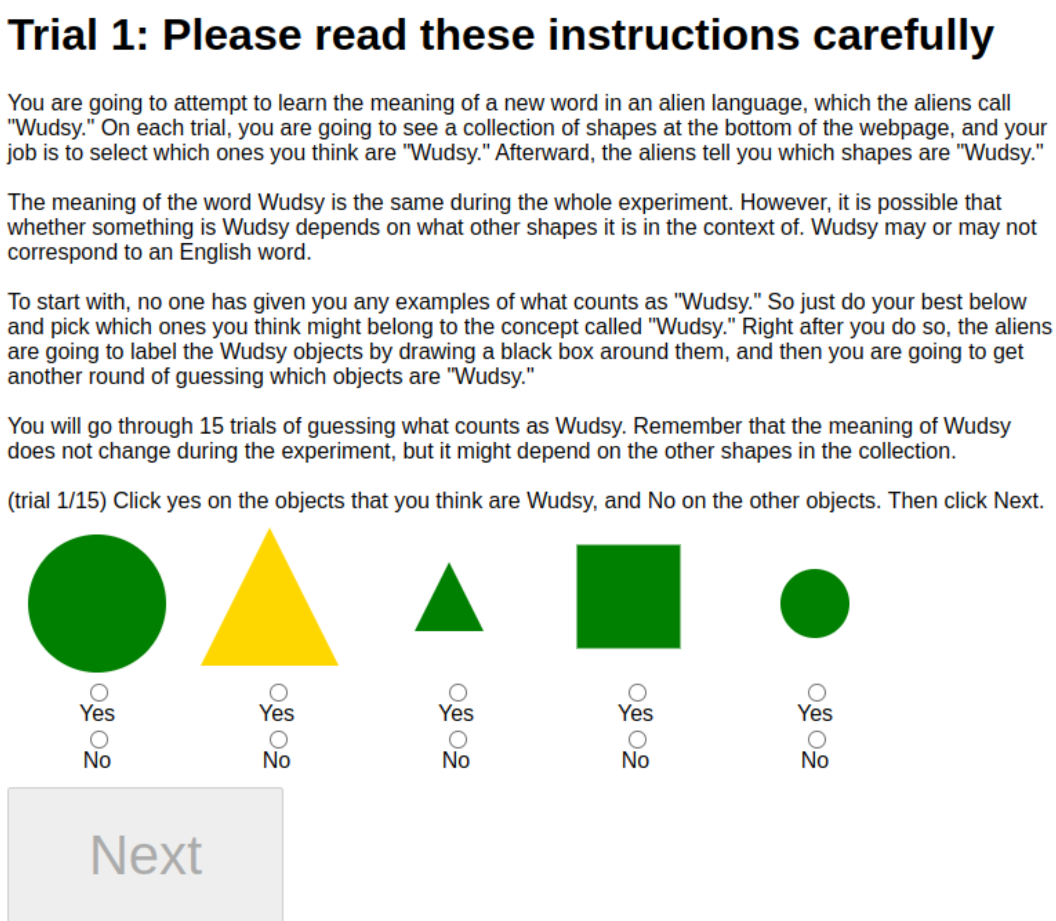

Figure 8: Cover story and web interface for our version of the logical concept learning study, which is based on [45]

A.3.2 Parameter fitting

Training consists of fitting the parameters T, θ (for the prior), ϵ (for the likelihood), α and β (for the logical concepts likelihood), and Platt transform parameters a, b (for the Number Game). In practice, this amounts to around 400 parameters, almost all of which come from θ.

We fit these parameters using Adam with a learning rate of 0.001. We perform 1000 epochs of training for the Number Game, and 100 epochs for logical concepts. There is a tenfold difference in the number of concepts, so this way they take about the same number of gradient steps.

For the number game we do 10-fold cross validation to calculate holdout predictions. For logical concepts we use the train-test split introduced in [45], which involves running different groups of human subjects on each concept twice, with different random examples. One sequence of random examples is arbitrarily designated as training data, and the other as holdout data.

All model were trained on a laptop using no GPUs. Training takes between a few minutes and an hour, depending on the domain and the model.

Some of the parameters that we fit, namely ϵ, α, β, cannot be negative. To enforce this we actually optimize the inverse logistic of those parameters.

A.3.3 MCMC over Logical Expressions

Fleet was used[1] to perform MCMC over logical expressions with the domain-specific primitives in this file, which include:

$$
\begin{aligned}
\text{true}, \text{false} &: \text{boolean} \\
\text{blue, yellow, green} &: \text{object} \to \text{boolean} \\
\text{rectangle, circle, triangle} &: \text{object} \to \text{boolean} \\
\text{small, medium, large} &: \text{object} \to \text{boolean} \\
\text{and, or, } \iff, \implies &: \text{boolean} \times \text{boolean} \to \text{boolean} \\
\forall, \exists &: (\text{shape} \to \text{boolean}) \times 2^{\text{object}} \to \text{boolean} \\
\text{filter} &: (\text{object} \to \text{boolean}) \times 2^{\text{object}} \to 2^{\text{object}} \\
\in &: \text{object} \times 2^{\text{object}} \to \text{boolean} \\
\iota &: 2^{\text{object}} \to \text{object} \cup \{\bot\}\text{, unique set element} \\
\text{empty} &: 2^{\text{object}} \to \text{boolean} \\
\text{same\_shape}, \text{same\_color}, \text{same\_size} &: \text{object} \times \text{object} \to \text{boolean} \\
\text{size<, size}\leq\text{, size>, size}\geq\text{,} &: \text{object} \times \text{object} \to \text{boolean}
\end{aligned}
$$

The model first constructed a large hypothesis space by performing MCMC for 1 minute per batch, and per learning curve. In one minute, Fleet makes approximately 10^6 MH proposals. There are a little more than 200 learning curves, and 25 batches per curve, for a total of about 5 billion MCMC proposals. In the main text, we abbreviate this analysis by referring to 10^9 proposals.

The top 10 samples per batch and per learning curve were retained. These top 10 samples samples were then deduplicated to yield 45 thousand hypotheses. Parameter fitting and posterior estimation was performed solely over those 45 thousand hypotheses.

Quantitatively, these are vastly more proposals than the models introduced in this paper. Quantitatively, these proposals are also derived in a very different way: the hypothesis space for the BPL learner is actually informed by data on other learning curves, and also on the same learning curve, but in the future batches.

It is in this sense that the BPL model is a computational-level theory, and not a process model, because human subjects could not be proposing hypotheses using data that is going to be seen in the future, or on other learning curves. However, the above strategy for proposing hypotheses is a very reasonable heuristic for constructing the support of the space of plausible logical hypotheses that a human learner might ever think of.

A.4 Prompting

A.4.1 Proposing hypotheses

For the number game we use the following prompt for code-davinci-002 to generate candidate concepts in natural language, given examples $X_{1:K}$. The example number concepts given in the prompt come from the cover score given to human participants [33]:

```
# Python 3
# Here are a few example number concepts:
# -- The number is even
# -- The number is between 30 and 45
# -- The number is a power of 3
# -- The number is less than 10
#
# Here are some random examples of numbers belonging to a different ↙
    ↳ number concept:
# X_{1:K}
```

[1] Running the model was graciously performed by the authors of [45], who provided us with the raw data.

```
# The above are examples of the following number concept:
# -- The number is
```

where $X_{1:K}$ is formatted by listing the numbers with a comma and a space between them.

For the number game we used the following prompt to generate candidate concepts in python (code baseline):

```
# Python 3
# Here are a few example number concepts:
# -- The number is even
# -- The number is between 30 and 45
# -- The number is a power of 3
# -- The number is less than 10
#
# Here are some random examples of numbers belonging to a different number concept:
# X_{1:K}
# Write a python function that returns true if ‘num‘ belongs to this number concept.
def check_if_in_concept(num):
   return
```

For logical concepts we used the following few-shot prompt for GPT-4 to generate candidate concepts:

```
Here three simple concepts, which specify when an object is 'positive' relative to an example collection of other objects. Before giving the rule for each concept, we give examples of collections of objects, and which objects in the collection are 'positive'.

Concept #1:
    An Example of Concept #1:
        POSITIVES: (big yellow rectangle)
        NEGATIVES: (big green circle), (medium yellow rectangle)
    Another Example of Concept #1:
        POSITIVES: (medium yellow rectangle)
        NEGATIVES: (big red circle), (small green circle)
Rule for Concept #1: Something is positive if it is the biggest yellow object in the example.


Concept #2:
    An Example of Concept #2:
        POSITIVES: (small yellow circle), (medium yellow rectangle)
        NEGATIVES: (big green circle), (big blue rectangle)
    Another Example of Concept #2:
        POSITIVES: (big blue circle), (medium blue rectangle)
        NEGATIVES: (small green circle), (medium yellow rectangle),
Rule for Concept #2: Something is positive if there is another object with the same color in the example.

Concept #3:
    An Example of Concept #3:
        POSITIVES: (small yellow circle), (medium yellow rectangle)
        NEGATIVES: (big green circle), (big blue rectangle)
    Another Example of Concept #3:
        POSITIVES: (small blue circle), (small blue triangle), (medium blue rectangle)
        NEGATIVES: (medium green triangle), (big yellow rectangle)
    Another Example of Concept #3:
        POSITIVES: (big red rectangle), (medium red rectangle), (big red triangle)
        NEGATIVES: (medium green triangle), (big yellow rectangle)
Rule for Concept #3: Something is positive if it is the same color as the smallest triangle in the example.
```

```
Now here are some examples of another concept called Concept #4, but this time we don't know the rule. Infer ten different possible rules, and make those ten rules as simple and general as you can. Your simple general rules might talk about shapes, colors, and sizes, and might make comparisons between these features within a single example, but it doesn't have to. Remember that a rule should say when something is positive, and should mention the other objects in the example, and should be consisting with what you see below.

Concept #4:
    X_{1:K}
Rule for Concept #4: Something is positive if...

Now make a numbered list of 10 possible rules for Concept #4. Start by writing "1. Something is positive if". End each line with a period.
```

Each sample from the above prompt generates 10 possible concepts formatted as a numbered list. We draw 10 times at temperature=1 to construct 100 hypotheses. To obtain fewer than 100 hypotheses we take hypotheses from each sampled list in round-robin fashion. We found that asking it to generate a list of hypotheses generated greater diversity without sacrificing quality, compared to repeatedly sampling a single hypothesis.

The above prompt provides in-context examples of first-order rules. We also tried using a different prompt for propositional concepts that illustrated the examples as a truth table, and gave in-context example rules that were propositional:

```
Here are some example concepts defined by a logical rule:

Rule: a triangle.
Rule: a green rectangle.
Rule: big or a rectangle (unless that rectangle is blue).
Rule: not both big and green.
Rule: either big or green, but not both.
Rule: either a rectangle or not yellow.
Rule: a circle.


Now please produce a logical rule for a new concept. Your rule should be consistent with the following examples. It must be true of all the positive examples, and not true of all the negative examples. The examples are organized into a table with one column for each feature (size, color, shape):

X_{1:K}

Please produce a simple rule that is consistent with the above table. Make your rule as SHORT, SIMPLE, and GENERAL as possible. Do NOT make it more complicated than it has to be, or refer to features that you absolutely do not have to refer to. Begin by writing "Rule: " and then the rule, followed by a period.
```

Using the first order prompt for every concept gives a R^2 = .80 fit to the human responses. Using both prompts gives the R^2 = .81 result in the main paper: the propositional prompt for the propositional problems, and the first order prompt for the higher order problems. We strongly suspect that a single prompt that just showed both propositional and higher-order in-context examples would work equally well, given that a single first-order prompt works about as well also, but we did not try that because of the high cost of using GPT-4.

On the first batch, the learner has not observed any examples. Therefore the above prompts do not apply, and we use a different prompt to construct an initial hypothesis space:

```
Here are some example concepts defined by a logical rule:

Rule: color is purple.
Rule: shape is not a hexagon.
Rule: color is purple and size is small.
Rule: size is tiny or shape is square.
Rule: size is not enormous.
Rule: color is red.

Please propose a some new concepts, defined by a logical rule. ↙
    ↳ These new concepts can only refer to the following features:
- shape: triangle, rectangle, circle
- color: green, blue, yellow
- size: small, medium, large

Please make your rules short and simple, and please write your ↙
    ↳ response on a single line that begins with the text "Rule: ". ↙
    ↳ Provide 100 possible rules.
```

We generate from the above prompt at temperature=0, and split on line breaks to obtain candidate rules.

A.4.2 Translating from natural language to Python

We translate Number Game concepts from English to Python via the following prompt for code-davinci-002, and generate at temperature=0 until linebreak:

```
# Write a python function to check if a number is C.
def check_number(num):
    return
```

We translate the logic cool concepts from English to Python using a series of in-context examples, again generating with temperature=0 until the text `#DONE` is produced.[2]

```
def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if ‘this_object‘ is positive according to the ↙
        ↳ following rule:
        Something is positive if it is not a small object, and not ↙
            ↳ a green object.
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # ‘this_object‘ is not a part of ‘other_objects‘
    # to get all of the examples, you can use ↙
        ↳ ‘all_example_objects‘, defined as ‘other_objects + ↙
        ↳ [this_object]‘
    # be careful as to whether you should be using ↙
        ↳ ‘all_example_objects‘ or ‘other_objects‘ in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it is not a small object, and not a ↙
        ↳ green object.
```

[2]This prompt is pretty long, and probably could be much shorter. Preliminary experiments suggested that a few in-context examples were very helpful, and so to increase the odds of the model working without much time spent prompt-engineering, we provided a large number of in-context examples.

```
    #START
    return (not this_size == 1) and (not this_color == "green")
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if `this_object` is positive according to the following rule:
        Something is positive if it is bigger than every other object
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use `all_example_objects`, defined as `other_objects + [this_object]`
    # be careful as to whether you should be using `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it is bigger than every other object
    #START
    return all( this_size > other_object[2] for other_object in other_objects )
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if `this_object` is positive according to the following rule:
        Something is positive if it is one of the largest
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use `all_example_objects`, defined as `other_objects + [this_object]`
    # be careful as to whether you should be using `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it is one of the largest
    #START
    return all( this_size >= other_object[2] for all_example_object in all_example_objects )
#DONE


def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)
```

```
    returns: True if ‘this_object‘ is positive according to the following rule:
        Something is positive if it is smaller than every yellow object
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # ‘this_object‘ is not a part of ‘other_objects‘
    # to get all of the examples, you can use ‘all_example_objects‘, defined as ‘other_objects + [this_object]‘
    # be careful as to whether you should be using ‘all_example_objects‘ or ‘other_objects‘ in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it is smaller than every yellow object
    #START
    return all( this_size < other_object[2] for other_object in other_objects if other_object[1] == "yellow" )
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if ‘this_object‘ is positive according to the following rule:
        Something is positive if there is another object with the same color
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # ‘this_object‘ is not a part of ‘other_objects‘
    # to get all of the examples, you can use ‘all_example_objects‘, defined as ‘other_objects + [this_object]‘
    # be careful as to whether you should be using ‘all_example_objects‘ or ‘other_objects‘ in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if there is another object with the same color
    #START
    return any( this_color == other_object[1] for other_object in other_objects )
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if ‘this_object‘ is positive according to the following rule:
        Something is positive if it has a unique combination of color and shape
    """
```

```
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use ↙
        ↳ `all_example_objects`, defined as `other_objects + ↙
        ↳ [this_object]`
    # be careful as to whether you should be using ↙
        ↳ `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it has a unique combination of color ↙
        ↳ and shape
    #START
    return all( this_shape != other_object[0] or this_color != ↙
        ↳ other_object[1] for other_object in other_objects )
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if `this_object` is positive according to the ↙
        ↳ following rule:
        Something is positive if it has the same color as the ↙
            ↳ majority of objects
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use ↙
        ↳ `all_example_objects`, defined as `other_objects + ↙
        ↳ [this_object]`
    # be careful as to whether you should be using ↙
        ↳ `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if it has the same color as the ↙
        ↳ majority of objects
    #START
    majority_color = max(["yellow", "green", "blue"], key=lambda ↙
        ↳ color: sum(1 for obj in all_example_objects if obj[1] == ↙
        ↳ color))
    return this_color == majority_color
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if `this_object` is positive according to the ↙
        ↳ following rule:
        Something is positive if there are at least two other ↙
            ↳ objects with the same shape
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
```

```
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use `all_example_objects`, defined as `other_objects + [this_object]`
    # be careful as to whether you should be using `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # Something is positive if there are at least two other objects with the same shape
    #START
    return sum(1 for other_object in other_objects if other_object[0] == this_shape) >= 2
#DONE

def check_object(this_object, other_objects):
    """
    this_object: a tuple of (shape, color, size)
    other_objects: a list of tuples of (shape, color, size)

    returns: True if `this_object` is positive according to the following rule:
        C
    """
    # shape: a string, either "circle", "rectangle", or "triangle"
    # color: a string, either "yellow", "green", or "blue"
    # size: an int, either 1 (small), 2 (medium), or 3 (large)
    this_shape, this_color, this_size = this_object

    # `this_object` is not a part of `other_objects`
    # to get all of the examples, you can use `all_example_objects`, defined as `other_objects + [this_object]`
    # be careful as to whether you should be using `all_example_objects` or `other_objects` in your code
    all_example_objects = other_objects + [this_object]

    # C
    #START
```

A.5 GPT-4 Baselines

Our GPT-4 baseline for each domain presented the examples $X_{1:K}$ in string form and then asked GPT-4 to respond Yes/No as to whether a test example X_{test} belonged to the same concept. GPT-4 was then queried at temperature=1 to collect 10 samples. Samples not beginning with 'y'/'n' were discarded, and the ratio of remaining samples that began with 'y' was computed (case insensitive).

We show below example prompts for the number and logic domains.

```
Here are a few example number concepts:
-- The number is even
-- The number is between 30 and 45
-- The number is a power of 3
-- The number is less than 10

Here are some random examples of numbers belonging to a possibly different number concept:
98, 81, 86, 93

Question: Does the number 42 belong to the same concept as the above numbers?
Answer (one word, yes/no):
```

Logical concept example prompt:

```
Here are some example concepts defined by a logical rule:

Rule for Concept #1: Something is positive if it is the biggest yellow object in the example
Rule for Concept #2: Something is positive if there is another object with the same color in the example
Rule for Concept #3: Something is positive if it is the same color as the smallest triangle in the example

Now please look at the following examples for a new logical rule.

    An Example of Concept #4:
        POSITIVES: none
        NEGATIVES: (large yellow circle), (small green circle), (medium green circle), (small yellow triangle)
    Another Example of Concept #4:
        POSITIVES: (small green circle), (large green circle)
        NEGATIVES: (large yellow circle), (medium blue circle)
    Another Example of Concept #4:
        POSITIVES: (small green rectangle)
        NEGATIVES: (medium yellow circle), (medium blue rectangle), (large green circle), (medium green circle)
    Another Example of Concept #4:
        POSITIVES: (medium green rectangle)
        NEGATIVES: (medium yellow circle), (small yellow rectangle), (medium yellow rectangle), (medium blue rectangle)
    Another Example of Concept #4:
        POSITIVES: (small green rectangle)
        NEGATIVES: (large yellow rectangle), (small yellow triangle), (medium green circle), (small blue rectangle)
    Another Example of Concept #4:
        POSITIVES: (medium green triangle)
        NEGATIVES: (medium blue triangle), (medium blue rectangle), (large blue triangle), (small yellow triangle)
    Another Example of Concept #4:
        POSITIVES: none
        NEGATIVES: (small yellow circle), (large blue circle)
    Another Example of Concept #4:
        POSITIVES: none
        NEGATIVES: (large green circle), (small blue rectangle), (small green triangle), (medium blue rectangle)
    Another Example of Concept #4:
        POSITIVES: (small green rectangle)
        NEGATIVES: (small yellow circle), (large blue rectangle)

Now we get a new collection of examples for Concept #4:
(medium blue triangle) (large yellow triangle) (small blue rectangle) (large blue circle) (small yellow circle)
Question: Based on the above example, is a (small yellow circle) in the concept?
Answer (one word, just write yes/no):
```

A.6 Latent Language Baseline

For fair comparison, we designed our latent language baseline to be as similar to our system as possible. It performs maximum likelihood estimation of a single concept, rather than estimate a full posterior, but uses the exact same prompts and likelihood functions as our model. The most important difference from the original latent language paper [22] is that instead of training our own neural models for language interpretation and language generation, we use pretrained models (Codex/code-davinci-002 and GPT-4).

A.7 Ablation of the proposal distribution

We ablate the proposal distribution by proposing hypotheses unconditioned on $X_{1:K}$. We accomplish this by drawing concepts from the following alternative prompt, which is designed to resemble the prompt used by the full model except that it does not include $X_{1:K}$:

```
# Python 3
# Here are a few example number concepts:
# -- The number is even
# -- The number is between 30 and 45
# -- The number is a power of 3
# -- The number is less than 10
# -- The number is
```

A.8 Pretrained prior

Our pretrained prior comes from the opensource model CodeGen [34], which was trained on source code. We chose this model because we suspected that pretraining on source code would give better density estimation for text describing precise rules. We formatted the rules as a natural language comment and prefixed it with a small amount of domain-specific text in order to prime the model to put probability mass on rules that correctly talk about numbers or shapes.

For the number game, we would query CodeGen for the probability of $p(C)$ via

```
# Here is an example number concept:
# The number is C
```

For the number game's code baseline, we would query CodeGen for the probability of $p(C)$ via

```
# Python 3
# Let's think of a number concept.
# Write a python function that returns true if `num` belongs to this number concept.
def check_if_in_concept(num):
    return C
```

For logical concepts we would query CodeGen for the probability of $p(C)$ via

```
# Here are some simple example shape concepts:
# 1. neither a triangle nor a green rectangle
# 2. not blue and large.
# 3. if it is large, then it must be yellow.
# 4. small and blue
# 5. either big or green.
# 6. C
```

Because the proposal distribution would generate rules beginning with the prefix "Something is positive if..." we would remove that text before computing $p(C)$ as above.