Towards General Natural Language Understanding with Probabilistic Worldbuilding

ABULHAIR SAPAROV

December 2021
CMU-ML-21-[TODO]

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee
TOM MITCHELL, Chair
WILLIAM COHEN
FRANK PFENNING
VIJAY SARASWAT

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Add dedication here.

ABSTRACT

We introduce the Probabilistic Worldbuilding Model (PWM), a new fully-symbolic Bayesian model of semantic parsing and reasoning, as a *first step* in a research program toward more domain- and task-general NLU and AI. Humans create internal mental models of their observations which greatly aid in their ability to understand and reason about a large variety of problems. In PWM, the meanings of sentences, acquired knowledge about the world, and intermediate proof steps in reasoning are all expressed in a unified human-readable formal language, with the design goal of interpretability. PWM is Bayesian, designed specifically to be able to generalize to new domains and tasks. We derive and implement an inference algorithm that reads sentences by parsing and abducing updates to its latent world model that capture the semantics of those sentences. We show that PWL is able to utilize acquired knowledge to resolve ambiguities during parsing, such as prepositional-phrase attachment, pronominal resolution, and lexical ambiguity, and is able to understand sentences with more complex semantics, such as definitions of new concepts. Additionally, we evaluate PWL on two out-of-domain question-answering datasets: (1) ProofWriter and (2) a new dataset we call FictionalGeoQA, designed to be more representative of real language but still simple enough to focus on evaluating reasoning ability, while being robust against heuristics. Our method outperforms baselines on both, thereby demonstrating its value as a proof-of-concept.

ACKNOWLEDGEMENTS

Put your acknowledgements here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was initially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

CONTENTS

LIST OF FIGURES

ACRONYMS

DRY Don't Repeat Yourself

API Application Programming Interface

UML Unified Modeling Language

1

INTRODUCTION

Despite recent progress in AI and NLP producing algorithms that perform well on a number of NLP tasks, it is still unclear how to move forward and develop algorithms that understand language as well as humans do. In particular, large-scale language models such as BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), GPT-3 (Brown et al., 2020), XLNet (Yang et al., 2019), and others were trained on a very large amount of text and can then be applied to perform many different NLP tasks after some fine-tuning. In the case of GPT-3, some tasks require very few additional training examples to achieve state-of-the-art performance. As a result of training on text from virtually every domain, these models are domain-general. This is in contrast with NLP algorithms that are largely trained on one or a small handful of domains, and as such, are not able to perform well on new domains outside of their training. Despite this focus on domain-generality, there are still a large number of tasks on which these large-scale language models perform poorly (Dunietz et al., 2020). Many such tasks require the ability reason, often with multiple hops, and/or with hard logical constraints such as negation. Other tasks require models to understand compositional semantics (Nie, Wang, and Bansal, 2019). These models often learn to rely on syntactic heuristics to avoid having to do semantic analysis or reasoning, and so they do not perform well on tasks that are more carefully crafted to be robust against such heuristics, such as HANS dataset for natural language inference (McCoy, Pavlick, and Linzen, 2019). As a consequence, these models often fail in real-world scenarios when presented with a challenge example from outside the dataset on which it was trained. The limitations of today's state-of-the-art methods become evident when comparing with the human ability to understand language (Bender and Koller, 2020; Gardner et al., 2019; Linzen, 2020; Tamari et al., 2020). Humans create rich mental models of the world from their observations which provide superior explainability, reasoning, and generalizability to new domains and tasks. These issues are not unique to NLP (Lake et al., 2016). How do we, as a field, get from today's state-of-the-art models to more general intelligence? What are the next steps to develop algorithms that can generalize to new tasks at the same level as humans? The lack of interpretability in many of these models makes these questions impossible to answer precisely. One promising direction is to change the evaluation metric: Brown et al. (2020), Linzen (2020), and many others have suggested *zero-shot* or *few-shot accuracy* to measure the performance of algorithms (i.e. the algorithm is evaluated with a new

dataset, wholly separate from its training; or in the case of few-shot learning, save for a few examples). While this shift is welcome, it alone will not solve the above issues.

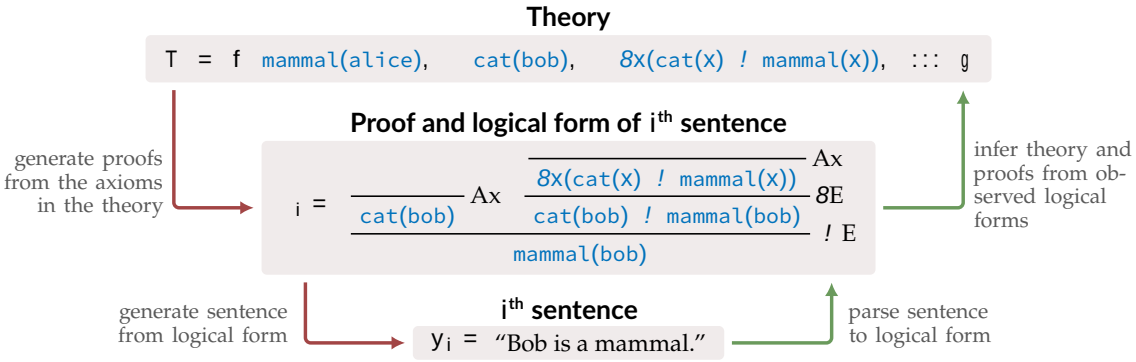


Figure 1: Schematic of the generative process and inference in our model, with an example of a theory, generating a proof of a logical form which itself generates the sentence "Bob is a mammal." During inference, only the sentences are observed, whereas the theory and proofs are latent. Given sentence y_i, the language module outputs the logical form. The reasoning module then infers the proof $_i$ of the logical form and updates the posterior of the theory T.

We introduce the *Probabilistic Worldbuilding Model* (PWM), a probabilistic generative model of reasoning and semantic parsing. Like some past approaches, PWM explicitly builds an internal mental model, which we call the *theory* (Charniak and Goldman, 1993; Hogan et al., 2020; Mitchell et al., 2018; Tamari et al., 2020). The theory constitutes what the algorithm believes to be true. PWM is fully symbolic and Bayesian, using a single unified human-readable formal language to represent all meaning, and is therefore *inherently interpretable* (i.e. every random variable in the model is well-defined with respect to other random variables and/or grounded primitives). Prior knowledge such as the rules of deductive inference, the structure of English grammar, knowledge of basic physics and mathematics can be incorporated by modifying the prior distributions of the random variables in PWM. Incorporating prior knowledge can greatly reduce the amount of training data required to achieve sufficient generalizability, and our experiments will demonstrate that PWM is a promising first step in this direction. Extensibility is key to future research that could enable more general NLU and AI, as it provides a clearer path forward for future exploration.

We present an implementation of inference under the proposed model, called *Probabilistic Worldbuilding from Language* (PWL). While PWM is an abstract mathematical description of the underlying distribution of axioms, proofs, logical forms, and sentences, PWL is the algorithm that reads sentences, computes logical form representations of their meaning, and updates the axioms and proofs in the theory accordingly. See figure 1 for a high-level schematic diagram of PWM and

PWL. PWM describes the process depicted by the red arrows, whereas PWL is the algorithm depicted by the green arrows. We emphasize that the reasoning in PWL is not a theorem prover and is not purely deductive. Instead, PWL solves a different problem of *abduction*, which in some ways is computationally easier than deductive inference: Given a set of observations, work backwards to find a set of axioms that deductively *explain* the observations. It is these abduced axioms that constitute the internal "mental model." Humans often rely on abductive reasoning, for example in commonsense reasoning (Bhagavatula et al., 2020; Furbach, Gordon, and Schon, 2015).

In machine learning, *generality* refers to the extent of an algorithm's ability to adapt to new domains or tasks outside of their training data. A core principle of our approach is *generality by design*. Simplifying assumptions often trade away generality for tractability, such as by restricting the representation of the meanings of sentences, or number of steps during reasoning. PWM is designed to be domain- and task-general, and to this end, uses higher-order logic (i.e. lambda calculus) (Church, 1940) as the formal language, which we believe is sufficiently expressive to capture the meaning of declarative and interrogative sentences in natural language. Furthermore, PWM uses *natural deduction* for reasoning, which is *complete* in that if a logical form is true, there is a proof of (Henkin, 1950). The priors should not be overly restrictive or domain-specific. But since the work in this paper represents a first step, we make simplifying assumptions to more quickly produce a proof-of-concept. For example, we assume that, given the theory, the logical forms and sentences are independent and identically distributed (i.i.d.). As a consequence, PWL is not able to parse cross-sentential anaphora (i.e. pronouns that refer to entities declared in other sentences). We also assume that the universe of discourse is constant. So for example, the sentence "All the kids are sleeping" would have the meaning of every child in the universe is sleeping, rather than the more likely meaning of every child in the current location is sleeping, though the universe can be specified explicitly such as in the example "All the kids in Pittsburgh are sleeping." Furthermore, we assume the language is noise-less (i.e. there are no spelling or grammar mistakes). Importantly, we make these assumptions in full awareness of the broader context of the more general model, leaving open a clear path for future research to relax those assumptions. Large and complex symbolic systems, such as the one we propose, can be difficult to initially design and implement while maintaining a high degree of generality, but this generality can help to avoid time-consuming redesign and reimplementation. A sufficiently well-designed and general algorithm does not need to be significantly adapted or retrained when applied to new tasks or domains. We do not claim to have achieved this ideal in this work, but it is a step in

the right direction, and is nevertheless a valuable property of more general algorithms.

We aim to test the following hypothesis in this thesis:

> *A knowledge-driven architecture can be designed and trained to understand individual sentences in documents from domains outside of which it was trained, construct a model of the world and use the knowledge therein to guide semantic parsing and resolve ambiguous interpretations, and compile and reason over the collection of acquired logical forms.*

The thesis statement can be dissected into four high-level claims:

1. *Generality.* PWL is able to read, understand, and reason over a large and diverse set of sentences and questions, with complex semantics, from domains of content beyond those its parser is trained on. PWL's use of abductive instead of deductive reasoning helps to ease the computational burden of this increased generality. The highly underspecified nature of the problem of abduction is alleviated by the probabilistic nature of PWL, as it provides a principled way to find the most probable theories. This ability includes, but is not limited to:
   - The ability to read, understand, and reason over sentences and questions from domains outside of the parser's training. For instance, figure 2 is an example from a geography question-answering dataset that we use to evaluate PWL, but the training set for the parser contains no sentences or proper nouns about geography. We emphasize that PWL only sees the context and question in the example at test-time, and no example from an evaluation dataset is included in the training set.
   - The ability to read, understand, and reason over sentences and questions with complex semantics, such as those that define new concepts. In the example given in figure 2, the sentence "Every river that is shorter than 400 kilometers is not major" declares a defining property of the concept "major," with which PWL must reason in order to answer the question. Another example of sentences with complex semantics are those that go beyond the expressivity of the Horn clause fragment of first-order logic, such as those involving classical negation.
2. *Exploit world model/acquired knowledge.* PWL is able to collate acquired knowledge into an internal model of the world, and utilize this knowledge when performing tasks. This includes, but is not limited to:
   - The ability to exploit background knowledge to resolve syntactic and semantic ambiguities during parsing, thereby im-

proving parsing accuracy. For example, the sentences "Sally caught a butterfly with a net" and "Sally caught a butterfly with a spot" have prepositional-phrase attachment ambiguity without any knowledge of butterflies, nets, spots, etc. We demonstrate that if PWL has previously read the sentences "No butterfly has a net" or "The butterfly has a spot," it is able to correctly resolve this ambiguity. The sentence "A butterfly has a spot and it is blue" contains pronominal resolution ambiguity: "it" can refer to either the butterfly or the spot. But if PWL has read the sentences "No red thing is blue" and "The spot is red," it is able to correctly resolve this ambiguity, as well. In another example, the word "largest" in "What is the largest state?" can refer to either largest in terms of area or population. We demonstrate that PWL can use knowledge acquired from previously-read sentences to resolve such lexical ambiguities, such as whether "largest city" refers to the city with the largest area or population.

- The ability to perform reasoning over the acquired knowledge, possibly with complex semantics, such as to answer the question in the example given in figure 2.

3. *Incorporate knowledge a priori.* We can capitalize on the large body of literature from fields such as linguistics, formal semantics, and proof theory, to provide stronger inductive biases in PWM, greatly improving its statistical efficiency. For instance, we are able to capitalize on the comprehensive set of English roots and inflections of Wiktionary and correctly parse previously unseen inflected forms of words, such as the superlative and comparative forms of adjectives, even if they are not present in the parser's training data. As a more concrete example, the parser's training data contains the example that "longest" refers to the property of maximizing length, and contains no examples with "long" or "longer." Wiktionary provides the comparative and superlative forms of "long," which PWM exploits to correctly parse "longer" as indicating that the length of one object is greater than that of another.

4. *Interpretability and extensibility.* All components of our proposed architecture are fully interpretable, in the sense that the meaning of every sentence and every fact in the theory is expressed in a well-founded human-readable formal language. Since PWM is Bayesian, any component or prior can be swapped out, extended, or composed with a richer or more sophisticated probabilistic model, and the resulting inference algorithm will guarantee the correct sharing of information across all components. Interpretability also helps researchers to determine which extensions are most useful and how to implement them in order

Context: "River Giffeleney is a river in Wulstershire. River Wulstershire is a river in the state of Wulstershire. River Elsuir is a river in Wulstershire. The length of River Giffeleney is 413 kilometers. The length of River Wulstershire is 830 kilometers. The length of River Elsuir is 207 kilometers. Every river that is shorter than 400 kilometers is not major."

Query: "What rivers in Wulstershire are not major?"

Figure 2: An example from FictionalGeoQA, a new fictional geography question-answering dataset that we created to evaluate reasoning in natural language understanding.

to achieve new functionality. For example, although initially designed to reason using *classical logic*, we were able to easily extend PWL to use *intuitionistic logic* in our experiments. As another example, a noise model would enable parsing and understanding of noisy text, and can be used suggest more accurate autocorrections, better informed by semantics and background knowledge.

1.1 RELATED WORK

Fully symbolic methods were commonplace in earlier AI research (Dreyfus, 1985; Newell and Simon, 1976). However, they were oftentimes quite brittle. All too often a new observation or input would contradict the internal theory or violate an assumption, and it was not clear how to resolve the contradiction in a principled manner and proceed. But they do have some key advantages: Symbolic approaches that use well-studied human-readable formal languages such as first-order logic, higher-order logic, type theory, etc. would enable humans to readily inspect and understand the internal processing of these algorithms, effecting a high degree of interpretability (Cooper et al., 2015; Dowty, 1981; Gregory, 2015). Symbolic systems can be made general by design, by using a sufficiently expressive formal language and ontology. Thus, hybrid methods have been explored to alleviate the brittleness of formal systems while engendering their strengths, such as interpretability and generalizability; for example, the recent work into *neuro-symbolic* methods (Saha et al., 2020; Tafjord, Dalvi, and Clark, 2021; Yi et al., 2020). Neural theorem provers are in this vein (Rocktäschel and Riedel, 2017). However, the proofs considered in these approaches are based on *backward chaining* (Russell and Norvig, 2010), which restricts the semantics to the Horn clause fragment of first-order logic. Arakelyan et al. (2021), Ren, Hu, and Leskovec (2020), and Sun et al. (2020) extend coverage to the existential positive fragment of first-order logic. In natural language, there are sentences that express more complex semantics such as including negation, nested universal

quantification, and higher-order structures. Kapanipathi et al. (2021) present a pipeline approach where a semantic parser is used in conjunction with a neuro-symbolic reasoning component (Riegel et al., 2020) to answer questions over a structured knowledge base. The reasoning component performs deductive reasoning over the function-free fragment of first-order logic, by reducing the problem of first-order reasoning into one of propositional reasoning. While there are provably efficient algorithms to perform deductive reasoning on these less expressive formal languages, our work explores the other side of the tradeoff between tractability and expressivity/generality. Theorem provers attempt to solve the problem of finding a proof of a given formula, from a given set of axioms. This is purely a problem of deductive reasoning. In contrast, the reasoning component of PWM is abductive, and the abduced axioms can be used in various downstream tasks, such as question-answering, and to better read new sentences in the context of the world model, as we will demonstrate. We posit that abduction is sufficient for more general natural language understanding. PWM combines Bayesian statistical machine learning with symbolic representations in order to handle uncertainty in a principled manner, "smoothing out" or "softening" the rigidity of a purely symbolic approach. In PWM, the internal theory is a random variable, and so if a new observation or input is inconsistent with one internal theory, there may be other theories in the probability space that are not inconsistent with the observation. The probabilistic approach provides a principled way to resolve these impasses.

PWM is certainly not the first to combine symbolic and probabilistic methods. There is a rich history of *inductive logic programming* (ILP) (Muggleton, 1991) and probabilistic ILP languages (Bellodi and Riguzzi, 2015; Cussens, 2001; Muggleton, 1996; Sato, Kameya, and Zhou, 2005). These languages could be used to learn a "theory" from a collection of observations, but they are typically restricted to learning rules in the form of first-order Horn clauses, for tractability. In natural language, it is easy to express semantics beyond the Horn clause fragment of first-order logic.

Knowledge bases (KBs) and *cognitive architectures* (Hogan et al., 2020; Kotseruba and Tsotsos, 2020; Laird, Newell, and Rosenbloom, 1987; Mitchell et al., 2018) have attempted to explicitly model domain-general knowledge in a form amenable to reasoning, which is a core component of the internal mental model that humans create. Cognitive architectures aim to more closely replicate human cognition. Some approaches use probabilistic methods to handle uncertainty (Jain et al., 2019; Niepert and Domingos, 2015; Niepert, Meilicke, and Stuckenschmidt, 2012). However, many of these approaches make strong simplifying assumptions that restrict the expressive power of the formal language that expresses facts in the KB. For example, many knowledge bases can be characterized as graphs, where each entity corresponds

to a vertex and every fact corresponds to a labeled edge. For example, the belief `plays_sport(serena_williams,tennis)` is representable as a directed edge connecting the vertex `serena_williams` to the vertex `tennis`, with the edge label `plays_sport`. While this assumption greatly aids tractability and scalability, allowing many problems in reasoning to be solved by graph algorithms, it greatly hinders expressivity and generality, and there are many kinds of knowledge that simply cannot be expressed and represented in such KBs. PWM does not make such restrictions on logical forms in the theory, allowing for richer semantics, such as definitions, universally-quantified statements, conditionals, etc.

1.2 OUTLINE

This thesis is organized as follows:

- In chapter 2, we present the high-level architecture of PWM and PWL. We describe the two principle components of the model, the language module and the reasoning module, and how they work together.
- In chapter 3, we present the reasoning module in greater detail. A precise mathematical description of the model is provided in section 3.2, including a discussion on the representation of the content of the theory and on the representation of the proofs. In section 3.3, we describe the algorithm that performs inference under this model, and the specifics of its implementation in PWL.
- In chapter 4, we present the language module in greater detail including implementation details about the training and parsing algorithms. In section 4.4, we apply this parsing approach to the GeoQuery and Jobs datasets (Tang and Mooney, 2000; Zelle and Mooney, 1996), using the Datalog representation of the provided logical form labels, and demonstrate that the accuracy of the parsed logical forms is comparable to that of the state-of-the-art on these datasets. Since the Datalog representation in these datasets are domain-specific, in section 4.5, we present a new wide-coverage semantic representation based on higher-order logic.
- In chapter 5, we provide qualitative and quantitative results on experiments that evaluate the capabilities of PWL end-to-end. In section 5.3, we apply PWL to the out-of-domain question-answering task in ProofWriter (Tafjord, Dalvi, and Clark, 2021) and achieve perfect zero-shot accuracy when using intuitionistic logic. However, since the sentences in ProofWriter are simple in structure, being automatically generated from templates, we create a new question-answering dataset called FictionalGeoQA,

consisting of marginally more syntactically-complex (but still overall simple) sentences. More importantly, the dataset is designed to be robust against algorithms that rely on simple heuristics to answer questions, and thus to more accurately measure their reasoning ability relative to other datasets. In section 5.4, we describe this dataset in further detail and show that PWL outperforms current state-of-the-art baselines.

- Finally, in chapter 6, we summarize the presented work and highlight the lessons learned. We discuss ways in which simplifying assumptions can be relaxed, perhaps with different design choices or with further research. For example, how can PWM be extended to feasibly handle cross-sentential anaphora? Discourse narrowing or broadening? Noise? We also review the advantages and disadvantages of the design choices in PWM and PWL, and consider alternatives that may work better in future work and implementations.

2 ARCHITECTURE OVERVIEW

In this chapter, we provide a high-level mathematical description of the *Probabilistic Worldbuilding Model* (PWM) and an overview of the implementation of inference under the proposed model, called *Probabilistic Worldbuilding from Language* (PWL). We describe the two principal components of PWM/PWL: the language module and the reasoning module, and how they work together. We also provide some background to Markov chain Monte Carlo methods, which are heavily employed by PWL in order to approximately compute intractable probabilities.

2.1 BACKGROUND: MARKOV CHAIN MONTE CARLO

PWM is a probabilistic model, and as with many other probabilistic models, there are many probabilities that we wish to compute, but are intractable to do so exactly. This is often the case in Bayesian statistical machine learning, where we aim to compute the posterior probability, conditioned on some observations. *Markov chain Monte Carlo* (MCMC) methods are a family of computational methods to sample from intractable probability distributions and to approximate intractable probabilities using these samples (Robert and Casella, 2004).

MARKOV CHAINS: A *Markov chain* is a sequence of random variables $z_1, z_2, \ldots$ with the *Markov property*: every variable z_i depends only on the previous variable in the sequence z_{i-1}. In addition, the conditional distribution of the next variable given the previous variable does not change. That is, the Markov chain is *homogeneous*: for all i, j, $p(z_i \mid z_{i-1}) = p(z_j \mid z_{j-1})$.

Let Z be the state space of each z_i ($z_i \in Z$ for all i). For simplicity, we will first consider the case where $Z = \{1, \ldots, n\}$ is finite. The conditional distribution can be written as a matrix $K \in \mathbb{R}^{n \times n}$:

$$p(z_i = a \mid z_{i-1} = b) = K_{ab}. \tag{1}$$

where $\sum_{a=1}^{n} K_{ab} = 1$ for all b, and $K_{ab} \in [0, 1]$ for all a and b. This matrix formulation is helpful for illustration since the distribution of each z_i can be written as a vector with length n, so $p(z_i) \in \mathbb{R}^n$, where each element is between 0 and 1, and their sum is 1. The distribution of the

next variable $p(z_{i+1})$ can be obtained via simple matrix multiplication:

$$p(z_{i+1}) = K \; p(z_i). \tag{2}$$

A Markov chain is called *irreducible* if for any starting value $s \in Z$, and any $a \in Z$, there exists a t such that $p(z_t = a \mid z_1 = s) > 0$. That is, any value a in the state space Z is reachable from any starting value s after a finite number of steps.

For a starting value $s \in Z$, the *return times* are the values of t such that $p(z_t = s \mid z_1 = s) > 0$. The *period* of the state s is the least common divisor of the set of return times at s. A Markov chain is called *aperiodic* if the period of every state is 1.

If a Markov chain is both irreducible and aperiodic, there exists a *stationary distribution* such that, for any starting value $s \in Z$,

$$\lim_{t \to \infty} p(z_t = a \mid z_1 = s) = \quad (a), \tag{3}$$

for all $a \in Z$. In the matrix formulation, K is an irreducible stochastic matrix, so it must have an eigenvalue of 1. is simply the eigenvector that corresponds to this eigenvalue: $= K$. A Markov chain is said to have *mixed* when its samples are "close" to the stationary distribution (i.e. when t is sufficiently large so that $p(z_t)$ is similar to).

Extending these results to the more general case where Z need not be finite or countable requires more finesse, but the intuition is the same. We refer the reader to Durrett (2010), Meyn and Tweedie (1993), and Robert and Casella (2004).

METROPOLIS-HASTINGS: The goal of MCMC is to sample from an intractable target probability distribution F, with the key idea being to construct a Markov chain such that its stationary distribution is the same as F. *Metropolis-Hastings* (MH) is one such method (Hastings, 1970). PWL uses MH to abduce the latent theory and proofs. At each step in the Markov chain z_i, MH *proposes* a change to the state z'. Then, MH computes the *acceptance probability*:

$$\min \quad 1, \frac{F(z')}{F(z_i)} \frac{p(z_i \mid z')}{p(z' \mid z_i)} \quad , \tag{4}$$

where $F(x)$ is the probability of x under the target distribution from which we wish to sample, $p(z' \mid z_i)$ is the probability of proposing the new state z' from the old state z_i, and $p(z_i \mid z')$ is the probability of the reverse of this proposal. MH then either accepts or rejects the proposed state according to the above acceptance probability. If MH accepts the proposal, then $z_{i+1} = z'$. Otherwise, $z_{i+1} = z_i$. In order to compute the above acceptance probability, it suffices to have an efficient algorithm to compute the ratio of probabilities $F(z')/F(z_i)$. And so if $F(\,)$ has an intractable normalization term, this term would not need

to be computed since it cancels in the ratio. It is not difficult to show that the stationary distribution of this Markov chain is indeed F.

A simple example of MH is to sample from the standard normal distribution $N(0,1)$. Start with the first sample at $z_1 = 0$. The proposal distribution is a uniform jump from the current position: $p(z' = t \mid z_i) = 1$ if $t \in [z_i - \frac{1}{2}, z_i + \frac{1}{2}]$, and $p(z' = t \mid z_i) = 0$ otherwise. The acceptance probability at each step is:

$$\min\left(1, \frac{\exp(-z'^2/2)}{\exp(-z_i^2/2)}\right) = \min\left(1, \exp\left(\frac{z_i^2 - z'^2}{2}\right)\right). \quad (5)$$

Given sufficiently many iterations i, the samples z_i will be distributed according to $N(0,1)$. Different choices of proposal distributions may affect the speed of this convergence, but so long as every measurable subset of R is reachable in a finite number of steps, convergence is guaranteed.

GIBBS SAMPLING: *Gibbs sampling* (Geman and Geman, 1984) is another MCMC method that is useful in high-dimensional settings. For example, PWL uses Gibbs sampling to learn the parameters of the parser. Suppose we have a probabilistic model containing k variables: $p(x_1, \ldots, x_k)$, where $x_i \in X_i$ for each i. The goal of Gibbs sampling is to sample from $p(x_1, \ldots, x_k)$. In MCMC, for this setting, each state in the Markov chain z_i can be written as a tuple: $z_i = (z_{i,1}, \ldots, z_{i,k})$, where each $z_{i,j}$ is identified with x_j, and so $Z = X_1 \times \cdots \times X_k$. The Gibbs sampling algorithm starts with initial values for $z_1 = (z_{1,1}, \ldots, z_{1,k})$. Then, for each iteration $i = 2, 3, \ldots$, iterate over each $j = 1, \ldots, k$, and sample $z_{i+1,j}$ from the conditional distribution

$$p(x_j \mid x_1 = z_{i+1,1}, \ldots, x_{j-1} = z_{i+1,j-1}, x_{j+1} = z_{i,j+1}, \ldots, x_k = z_{i,k}), \quad (6)$$

which is the distribution where all variables other than x_j are fixed to their current values in the Markov chain. With sufficiently many iterations, the samples $(z_{i,1}, \ldots, z_{i,k})$ will be distributed according to $p(x_1, \ldots, x_n)$. Gibbs sampling is an attractive option when the above conditional distribution is easy to sample. Interestingly, Gibbs sampling can be shown to be an instance of Metropolis-Hastings where the acceptance probability is always 1.

2.2 PROBABILISTIC WORLDBUILDING MODEL

PWM is a probabilistic generative model of sentences that aims to capture a central aspect of human language understanding: a rich mental model of the world that is constructed and maintained from those observed sentences. In PWM, this internal mental model is called the *theory* and is a collection of logical forms. PWM describes the conditional distribution of natural language sentences, given the theory.

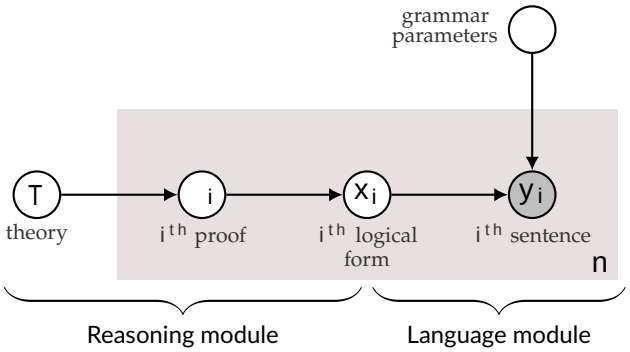


Figure 3: Graphical model representation of PWM. Shaded nodes indicate observed random variables, whereas unshaded nodes indicate unobserved (i.e. latent) random variables. T, $_i$, and are each composed of many random variables, but they are omitted here for illustration.

More precisely, PWM is the collection of random variables (T, , x, y,) and conditional distributions, where T is the *theory*, , $f_1, \ldots,$ $_n g$ are the *proofs* of each logical form, x , $fx_1, \ldots, x_n g$ are the *logical forms* of each observation, y , $fy_1, \ldots, y_n g$ are the observed *sentences*, and are *grammar parameters* that control the conditional distribution of the sentences given the logical form $p(y_i \mid x_i,$). The prior and conditional distributions in PWM are: p(T), p( ), $p($ $_i \mid T)$, $p(y_i \mid x_i,$). The logical form x_i is the conclusion of the proof $_i$, and so x_i is deterministic given $_i$. These conditional and prior distributions define a joint distribution over all the variables in the model. At a high level, the process for generating a sentence sampled from this probability distribution is:

1. Sample the theory T from a prior distribution p(T). T is a collection of logical forms in higher-order logic that represent what PWL believes to be true.

2. Sample the grammar parameters from a prior distribution p( ).

3. For each observation i, sample a proof $_i$ from $p($ $_i \mid T)$. The conclusion of the proof is the logical form x_i, which represents the meaning of the i^{th} sentence.

4. Sample the i^{th} sentence y_i from $p(y_i \mid x_i,$).

Inference is effectively the inverse of this process, and is implemented by PWL. During inference, PWL is given a collection of observed sentences y and the goal is to discern the value of the latent variables: the logical forms for each sentence x, the proofs for each logical form , and the underlying theory T. Both the generative process and inference algorithm naturally divide into two modules:

- **Language module:** During inference, this module's purpose is to infer the logical form of each observed sentence. That is, given

the input sentence y_i, this module outputs the k most-probable values of the logical form x_i (i.e. semantic parsing).

- **Reasoning module:** During inference, this module's purpose is to infer the underlying theory that logically entails the observed logical forms (and their proofs thereof). That is, given an input collection of logical forms $x_1, \dots, x_n$, this module outputs the posterior distribution of the underlying theory T and the proofs $_1, \dots, _n$ of those logical forms.

Note that the y_i need not necessarily be sentences, and PWM can easily be generalized to other kinds of data. For example, if a generative model of images is available for $p(y_i \mid x_i)$, then an equivalent "vision module" may be defined. This vision module may be used either in place of, or together with the language module, and would provide a principled way to combine information from multiple modalities. In the above generative process, PWM assumes each sentence to be independent. A model of context is required to properly handle inter-sentential anaphora or conversational settings. This can be done by allowing the distribution on y_i to depend on previous logical forms or sentences: $p(y_i \mid x_1, \dots, x_i)$ (i.e. relaxing the i.i.d. assumption). Figure 3 provides a graphical representation of PWM.

2.3 PROBABILISTIC WORLDBUILDING FROM LANGUAGE

In our proposed model, *reading* is the act of updating the posterior distribution of the theory, given a new sentence. We derive and implement an algorithm called *Probabilistic Worldbuilding from Language* (PWL) which performs this posterior inference in order to read and understand sentences. PWL also uses the inferred theory for downstream tasks, such as answering questions about the sentences. The posterior of the model is given by

$$p(T, \ , \ , x \mid y) \ / \ p(T)p(\ ) \prod_{i=1}^{n} p(\ _i \mid T)p(y_i \mid x_i, \ ). \tag{7}$$

We are able to exploit the structure of the posterior distribution to make the inference algorithm both simpler and faster. For instance, the posterior distribution of the grammar parameters has very little uncertainty, given that n is not too small. Thus we can split the inference algorithm into three "procedures":

1. *Training the parser:* Given a seed training set of labeled sentences x and y, learn the grammar parameters by computing their posterior $p(\mid x, y)$. PWL uses Gibbs sampling to obtain samples from this posterior (see section 4.3.1 for further details). These samples of are necessary to parse new sentences. Figure 4 provides an example from the seed training set. Our training algorithm

Sentence: "Which inner planet has the highest mass?"

Logical form:
```
z. X(X= x( i(inner(i)   arg1_of(x)=i)   planet(x))
   X(z)   f((f= x. v. m( y(value(y)   arg2(y)=v   arg1_of(m)=y)
   mass(m)   h(arg1(h)=x   has(h)   present(h)   arg2(h)=m)))
    g(greatest(f)(g)   arg1(g)=X   arg2(g)=z)))
```
(i.e. what is the value of z such that there exists a set of inner planets X, z is a member of X, and there exists a function f that returns the mass of its input, such that z maximizes f over the set X)

Derivation tree:

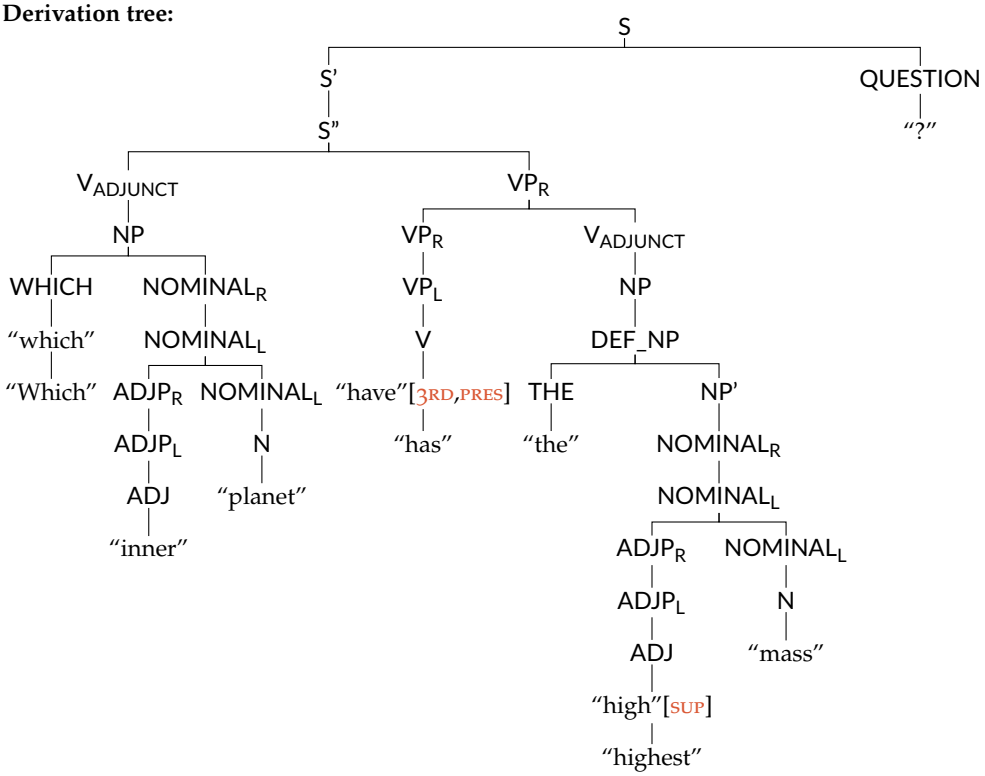


Figure 4: An example from the seed training set of PWL, labeled with the logical form and derivation tree (i.e. syntax tree). This example helps to train the parser in PWL. The semantic formalism for the logical form is detailed in section 4.5. Details on the grammar model of the language module are provided in section 4.2. Note that the derivation tree label is not necessary, as we will provide a training algorithm in section 4.3.1 that only uses sentences with logical form labels, and semantic parsing experiments in section 4.4 where derivation tree labels are not included in training. In the above example derivation tree, 3RD is a morphological flag that indicates the third person, PRES indicates present tense, and SUP indicates superlative. Semantic transformation functions are omitted for brevity.

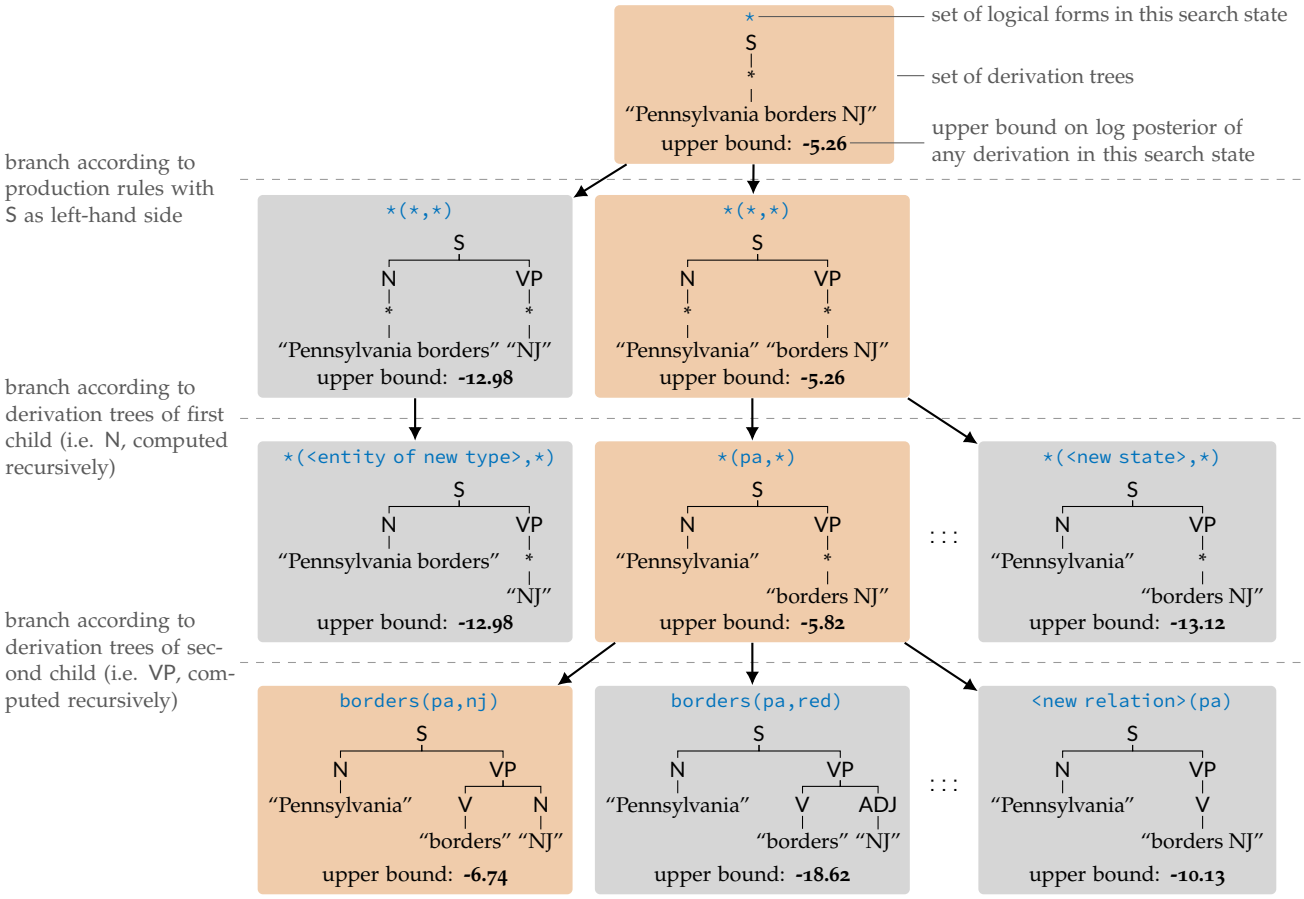


Figure 5: The search tree of the branch-and-bound algorithm during the parsing of the sentence "Pennsylvania borders NJ." In this diagram, each block is a search state, which represents a set of logical forms and derivation trees. The blue asterisk ⋆ denotes the set of all possible logical forms, whereas the black asterisk * denotes the set of all possible derivation (sub)trees. The gray-colored search states are unvisited by the parser, since their upper bounds on the objective function are smaller than that of the completed parse at the bottom of the diagram (-6.74), thus allowing the parser to ignore a very large number of improbable logical forms and derivations. For simplicity of illustration, the example here uses a simplified grammar and logical formalism, and the branching steps are simplified. The recursive optimization of the derivation subtrees for N and VP are not shown, which have their own respective search trees. This algorithm is described in greater detail in section 4.3.2.

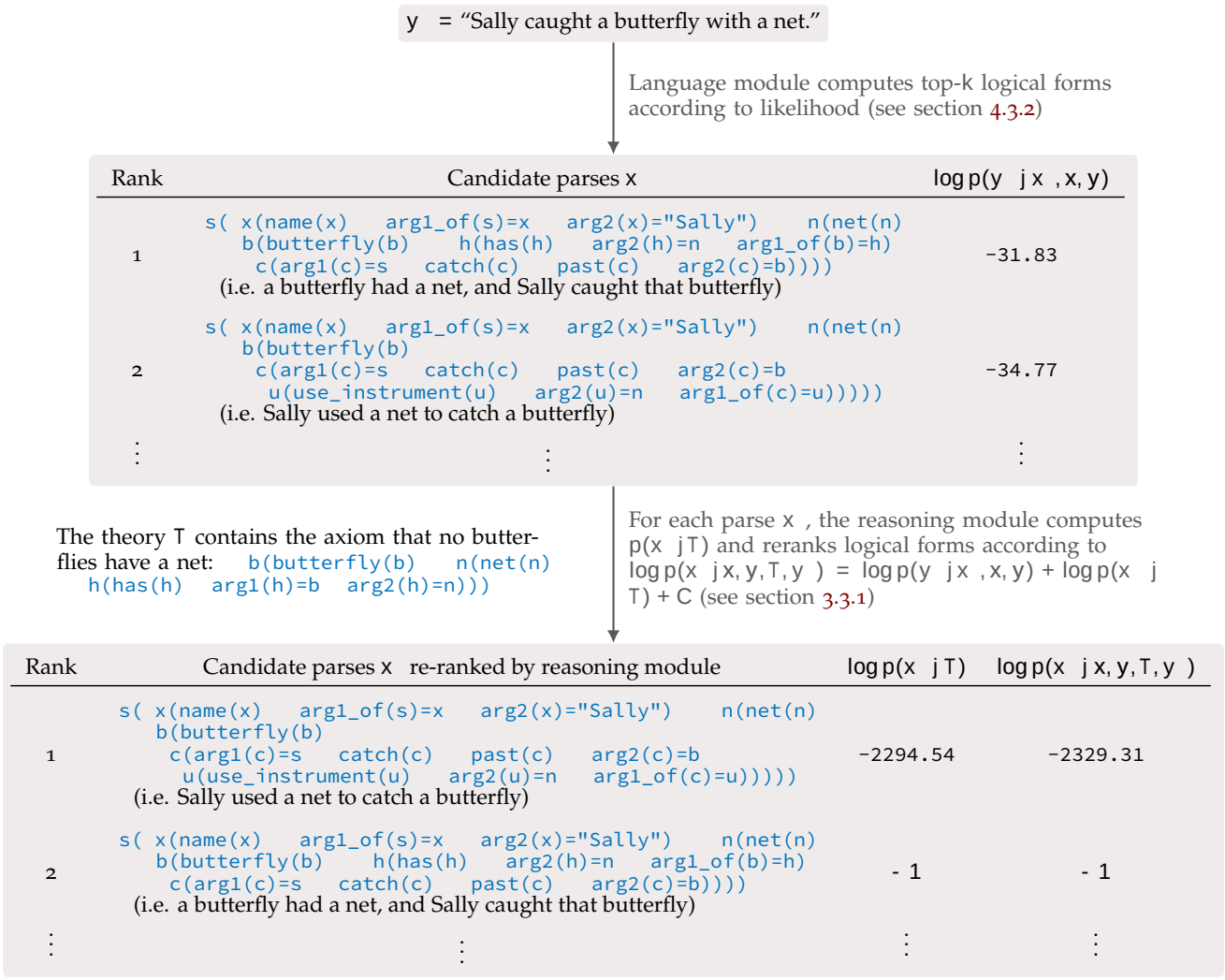


Figure 6: An example where PWL reads the sentence "Sally caught a butterfly with a net," which is a classical example of a sentence with prepositional phrase attachment ambiguity: "with a net" could either attach to "butterfly" or "caught." In PWL, "reading" a sentence is divided into two stages: (1) find the k most likely logical forms, ignoring the prior probability of each logical form conditioned on the theory, and (2) for each logical form in the list, computing its prior probability conditioned on the theory and then re-ranking the list accordingly. The output of the first stage is shown in the top table, and the output of the second stage is shown in the bottom table. In this example, the theory contains the axiom that no butterflies have a net, which itself is the result of having read the sentence "No butterfly has a net." The log probabilities in the bottom table are unnormalized. The semantic formalism for the logical forms is detailed in section 4.5.

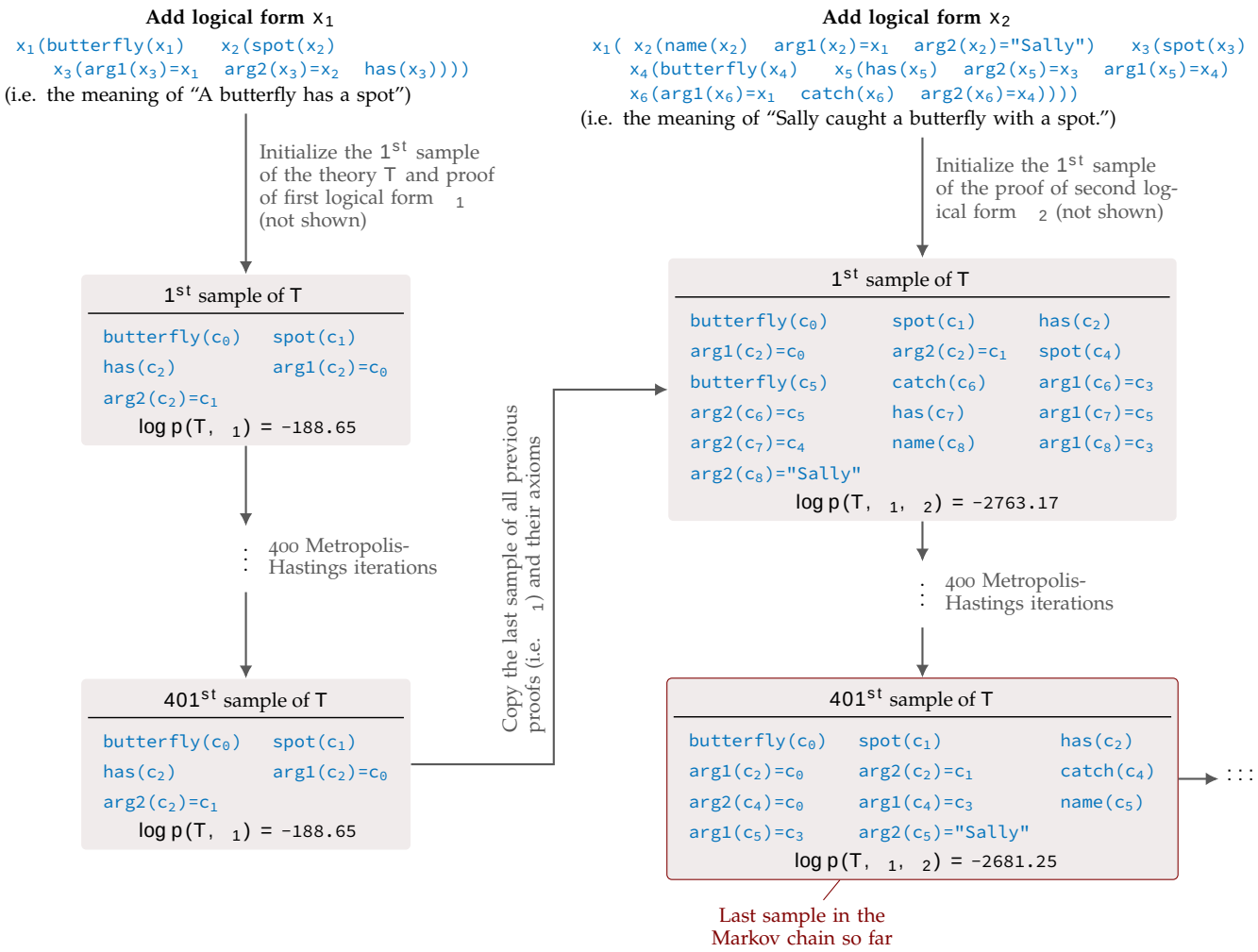


Figure 7: An example of PWL abducing the posterior distribution of the theory given two logical forms: x_1 is the meaning of "A butterfly has a spot," and x_2 is the meaning of "Sally caught a butterfly with a spot." PWL does not represent the full posterior distribution, but rather it keeps samples of the Markov chain that serve as an approximation of the posterior. Additional samples from the posterior can be produced by performing additional Metropolis-Hastings iterations, starting from the last sample. The proofs for each logical form $_1$ and $_2$ are not shown here for brevity, but $_1$ is shown in figure 8. Note that after adding the logical form x_2 in the top-right of the figure (the meaning of "Sally caught a butterfly with a spot"), the theory contains many more axioms. For example, there are two butterflies: c_0 is the butterfly with a spot, and c_5 is the butterfly that Sally caught. But since the prior distribution of the theory T favors smaller and more parsimonious theories, and Metropolis-Hastings tends to visit samples with higher and higher probabilities, then after 400 iterations, these two butterflies were merged into a single entity c_0, thus simplifying the overall theory. Figure 8 shows a sample of $_1$, which is the abduced proof of the first logical form. Section 3.3 describes this algorithm in more detail.

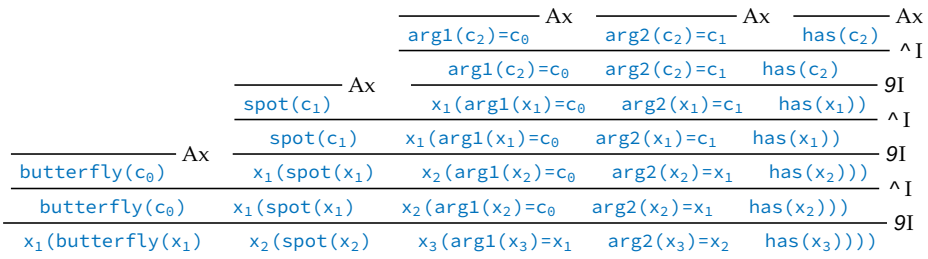


Figure 8: The "theory abduction" procedure in PWL takes as input the logical form shown at the bottom of this figure, which has the meaning of "A butterfly has a spot." The output of this procedure is the abduced proof of this logical form shown here, whose axioms constitute the abduced theory (labeled "Ax"). The example in figure 7 shows the output abduced theory, where x_1 is the logical form shown here in figure 8 and the above proof is $_1$. Each horizontal bar denotes a proof step. Steps labeled "Ax" are axioms. These are the axioms that constitute the theory T, and are shown in the samples of T in figure 7. The steps labeled "^I" are conjunction introduction steps, where if we are given that A and B are true, we conclude that A ^ B is true. The steps labeled "9I" are existential introduction steps, where if we are given that [x 7! c] is true where x is a variable and c is a symbol and [x 7! c] is the formula where x is substituted with c, then we can conclude that 9x. is true. The conclusion of the proof is the logical form for the sentence "A butterfly has a spot."

can additionally accept derivation tree labels (i.e. syntax trees) for some or all of the sentences in the training set. These derivation tree labels are not necessary, as our algorithm described in section 4.3.1 is able to train the parser without them, but they can help during debugging. Another way to train the parser would be to compute the *maximum a posteriori* (MAP) estimate max p(j x, y), but this is intractable in our model, and so we choose to sample from the posterior instead. But in extended or modified versions of PWM where the parsing model is different, MAP could be an attractive method to train the parser.

2. *Parsing:* Given a new (unseen) sentence y , find the k most probable values of the logical form x , according to the posterior distribution

$$p(x \mid x, y, y, T) \propto p(x \mid T)p(y \mid x, x, y).$$

However, since the computation of $p(x \mid T)$ is highly non-trivial, we divide it into two stages: First, find the k-best parses according to the likelihood

$$p(y \mid x, \ldots, x, y) = \int p(y \mid x, \ldots, \ldots)\, p(\ldots \mid x, y)\, d\ldots \approx \frac{1}{N_{\text{samples}}} \sum_{\ldots^{(t)} \mid \mathbf{x}, \mathbf{y}} p(y \mid x, \ldots, \ldots^{(t)}).$$

This is a discrete optimization problem, which PWL solves using *branch-and-bound* (Land and Doig, 1960). This algorithm begins by considering the set of all possible logical forms (and derivation trees). Next, it divides this set into a number of disjoint subsets (the "branch" step), and for each subset, it computes an upper bound on the objective function over any logical form in that subset (the "bound" step). Each subset is pushed onto a priority queue, where the priority is the upper bound. Then the algorithm pops the highest priority subset and repeats this process: further subdividing the set into subsets, computing the upper bound for each, and pushing them onto the priority queue. The algorithm repeats until it finds a subset containing a single logical form, whose likelihood is at least the highest priority of the priority queue. This logical form is guaranteed to be optimal. The algorithm continues until the k-best logical forms have been found. Section 4.3.2 will go into more detail on this procedure. An example of this algorithm finding the most likely logical form for the sentence y = "Pennsylvania borders NJ" is shown in figure 5. Note that the grammar and logical formalism are simplified in this example.

Once the k most likely logical forms are computed, in the second stage, PWL then re-ranks the resulting logical forms according to the semantic prior $p(x \mid T)$. Figure 6 shows the result of PWL parsing the sentence y = "Sally caught a butterfly with a net," where the prepositional phrase attachment ambiguity is resolved using background knowledge. The parsing procedure is depicted as the first green arrow in figure 1 going from the sentence to the logical form.

3. *Theory abduction:* Given a collection of logical forms $x_1, \ldots, x_n$, abduce a proof $\ldots_i$ for each logical form x_i, with the axioms of the proofs constituting the abduced theory T. This abduction is done by computing the posterior distribution of the theory and proofs $p(T, \ldots_1, \ldots, \ldots_n \mid x_1, \ldots, x_n)$. Since computing this posterior exactly is intractable, we approximate it using Metropolis-Hastings (MH) to produce posterior samples of the theory T and all proofs $\ldots_1, \ldots, \ldots_n$. This inference is done in a *streaming* fashion: Given a new logical form x_{n+1}, initialize the MH using the previous

samples of T and $_1, \ldots, _n$; Then, continue MH to produce posterior samples of T and $_1, \ldots, _n, _{n+1}$. Figure 7 shows the process of PWL abducing a theory for two logical forms: x_1 is the logical form meaning of "A butterfly has a spot," and x_2 is the logical form meaning of "Sally caught a butterfly with a spot." Furthermore, this algorithm can provide estimates of the probabilities of logical forms $p(x_i \mid T)$, which is useful for tasks such as question-answering, as well as for computing the semantic prior for parsing. Note that seed axioms may be added directly to the theory here. Section 3.3 will go into more detail on this procedure. This step is depicted as the second green arrow in figure 1 going from the logical form and proof to the theory.

Note that the first two procedures (*training the parser* and *parsing*) are governed by the language module whereas the third procedure (*theory abduction*) is governed by the reasoning module.

Normally in Bayesian inference, we would need to compute the posterior for *all* latent random variables in the model, including the logical forms x. However, it isn't obvious how to compute the conditional $p(x_i \mid T, y_i,)$. In addition, natural languages have evolved to express large quantities of information with minimal energy expenditure, in order to provide an efficient means of communication. As a result, in order to maximize information, the semantic content of sentences tend to be unambiguous, given the appropriate context and background knowledge. As a result, the posterior distribution $p(x_i \mid y_i)$ of a logical form interpretation x_i given a sentence y_i is usually concentrated at one or a small handful of logical forms (modes). To exploit this fact and to simplify the implementation of the language module, during inference, after parsing each sentence y_i into its logical form x_i, we assume the logical forms are fixed. But in general, there may exist real-world scenarios in which the meaning of some sentences are ambiguous, even in consideration of the context and background information, and their logical form interpretations should be allowed to vary.

Only the language module has latent parameters that need to be learned (i.e.). The model of the theory and proofs does not have any random variable parameters that need to be learned, and so the reasoning module does not need to be trained. But axioms can be added to the theory apriori, such as domain-independent facts about the world. The language module of PWL is trained with a seed training set consisting of a collection of labeled sentences, which are only used to train the parser. The seed training set also consists of two seed axioms which are added directly to the theory.

2.4 KEY DESIGN CHOICES

The key design choices discussed in this chapter are:

- We designed PWM as a probabilistic generative model of language understanding and reasoning. The *theory* is a random variable in PWM which attempts to explicitly encapsulate the human ability to create rich mental models of the world from their observations. We believe this sort of task-, modality-, and domain-independent model of the world is instrumental for further progress in AI.

- The probabilistic nature of PWM both helps to alleviate both the brittleness of fully symbolic systems which plagued earlier efforts in AI, as well as the highly underspecified nature of the problem of abduction. The number of possible explanations for a collection of observations is extremely large, and the ability to assign a probability to each explanation helps to focus the search on higher-probability theories.

- PWM uses higher-order logic, a symbolic human-readable formal language, to represent all knowledge in the theory as well as the meanings of sentences. This greatly aids in the interpretability of PWL, and enables it to utilize well-studied methods for reasoning in higher-order logic. Full higher-order logic is highly expressive and is able to represent the meaning of a very broad set of natural language sentences and phrases, independent of their domain.

- Though the generative process of PWL is one of deductive reasoning, the inference (implemented by PWL) performs abductive reasoning, which is computationally easier than deductive reasoning since PWL can add axioms as needed in order to find a proof for an observed logical form. This helps to workaround some of the issues of decideability in deductive reasoning.

- PWM is a Bayesian model, so every random variable in the model has a prior distribution. These priors enable us, as the designers, to incorporate background/expert knowledge to improve the statistical efficiency of the model. For example, we will show that providing some of the grammar rules for English syntax greatly improves the statistical efficiency of the language module.

- PWL is divided into two modular components: (1) the reasoning module, and (2) the language module. This helps streamline the implementation and debugging of the system, and by keeping the reasoning module independent of the perceptual modality, keeps open the path for future work to add new perceptual modules.

- Each sentence is assumed to be context-independent: conditioned on the theory, the sentences are independent and identically distributed. This is a strong assumption, and in order to

relax it, PWM must be extended to include a proper model of context, so that each sentence is no longer conditionally independent of the previous sentences given the theory. We will discuss potential ways to do so later in section 4.7.

- The procedure for reading sentences is divided into two stages: (1) first maximize the likelihood and find the k-best logical forms, then (2) compute the semantic prior for each logical form and rerank the list according to the posterior. We chose this approach since the computation of the semantic prior is non-trivial and computationally expensive.

- During inference, once PWL has computed the most probable logical form for a given sentence, that logical form is fixed and is not allowed to vary (i.e. the approximate posterior for the logical form of each sentence is a point estimate). This simplifies the implementation of the language module and works sufficiently well in our experiments, there are scenarios in which the meanings of sentences are ambiguous, even in considering of the context and background information. Thus, future work to allow the logical form to vary during inference would be valuable.

- In theory abduction, the reasoning module aims to approximate the posterior distribution p(T, j x) of the theory T and proofs given the observed logical forms x. We chose to infer the full posterior rather than a point estimate since natural language is not unambiguous, and real-world observations often have multiple probable explanations. Human language understanding and generation also preserves information about uncertainty, which is evident from the existence and ubiquity of words such as "probably," "maybe," "could," etc.

- However, even though T is a random variable, each individual sample of T is deterministic. But this would imply that words such as "probably," "maybe," and "could" would never be generated. This is not an issue in this thesis since none of the experiments have sentences that express uncertainty. But to correctly understand the meaning of these words, PWM needs to be extended so that each individual sample of T is probabilistic.

- The reasoning module uses Metropolis-Hastings (MH) to perform theory abduction. As with any MCMC method, MH has the desirable property that as more time (i.e. iterations) is spent performing inference, the better the approximate theory, becoming exact in the limit. MH is able to sample from distributions that have an intractable normalization term, since the normalization term cancels in the acceptance probability (equation 4). In PWL, the posterior of the theory and proofs conditioned on the

observed logical forms $p(T, \; | \; x)$ is one such distribution. This property is not unique to MH among MCMC methods.

- The reasoning module performs inference in a streaming fashion. This choice derives from the observation that when humans read sentences, each new sentence is unlikely to change the entire world model in the mind of the reader. Rather, each sentence provides more of an incremental update to the reader's beliefs. In PWL, this serves to provide a better starting point for MH during theory abduction, and as a result, reduce the number of iterations needed to find a good approximate theory.

We will describe the reasoning module in much greater detail in chapter 3, and the language module and its training in chapter 4.

3

REASONING MODULE

In this chapter, we present the reasoning module in greater detail. This module governs the theory as well as the proofs of the logical form observations. A mathematical description of the model is provided in section 3.2, including a discussion on the representation of the content of the theory and on the representation of the proofs. In section 3.3, we describe the algorithm that performs inference under this model, and the specifics of its implementation in PWL.

3.1 BACKGROUND: DIRICHLET PROCESSES

Before introducing the model for the reasoning module in the next section, we present background on Dirichlet processes, which forms a component in the prior distribution of the theory in PWM.

The *Dirichlet process* (DP) (Ferguson, 1973) is a distribution over probability distributions (i.e. samples from a DP are themselves distributions). If a distribution G is drawn from a DP, we can write

$$G \quad \mathrm{DP}(\ , H), \tag{8}$$

where the DP is characterized by two parameters: a concentration parameter $ > 0$ and a base distribution H. The DP has the useful property that $E[G] = H$, and the concentration parameter $$ describes the "closeness" of G to the base distribution H. If $$ is small, G is more different from the base distribution H. If $$ is large, G is more similar to H.

DPs are often used in statistical machine learning models where observations $y_1, y_2, \ldots$ are distributed according to G, such as in:

$$G \quad \mathrm{DP}(\ , H), \tag{9}$$

$$y_1, y_2, \ldots \quad G. \tag{10}$$

The joint probability of $y_1, \ldots, y_n$ is given by:

$$p(y_1, \ldots, y_n) = \frac{^{n} (\ )}{(\ + n)} \prod_{k=1}^{m} H(y_k) \ (n_k), \tag{11}$$

where y_k are the unique values of $y_1, \ldots, y_n$, m is the number of such values, $n_k \ , \ \#\{i : y_i = y_k\}$ is the number of times y_k appears in $y_1, \ldots, y_n$, and $^{n} ( ) = (\ + n)$ is the normalization term.

In these models, the *Chinese restaurant process* (CRP) (Aldous, 1985) provides a convenient equivalent description:

$$_1, \ _2, \ldots \quad H, \tag{12}$$

$$z_1 = 1, \tag{13}$$

$$z_{i+1} = \begin{cases} k & \text{with probability } \frac{n_k}{+i}, \\ k^{new} & \text{with probability } \frac{}{+i}, \end{cases} \tag{14}$$

$$y_i = \ _{z_i}, \tag{15}$$

where $n_k \ , \ \#\{j \ 6 \ i : z_j = k\}$ is the number of times k appears in $\{z_1, \ldots, z_i\}$, $k^{new} \ , \ \max\{z_1, \ldots, z_i\} + 1$ is the next integer that doesn't appear in $\{z_1, \ldots, z_i\}$. The analogy to a restaurant is to imagine a restaurant with a countably infinite sequence of tables, labeled $1, 2, 3, \ldots$ The first person comes into the restaurant and sits at table 1. For each subsequent person that enters the restaurant, they choose to sit at a table with probability proportional to the number of people already sitting at that table. Otherwise, they choose to sit at an empty table with probability proportional to $\ . \ z_i$ indicates which table the i^{th} customer chose to sit, n_k is the number of people sitting at table k, and k^{new} is the index of the next unoccupied table. Each table is assigned a sample from H, independently and identically distributed (i.i.d.), where $_i$ is the sample assigned to table i. Each observation y_i is the sample from H that is assigned to the table that the i^{th} customer chose to sit (i.e. table z_i). The CRP provides a simple algorithm to generate samples from a DP model. Notice that if $\ $ is very large, every customer is likely to choose to sit at a new table, and so each y_i is likely to be drawn i.i.d. from H (and therefore, G would be very similar to H). The opposite would be true in the case where $\ $ is small, where G would be heavily concentrated on a small handful of observations, as each customer is more likely to sit at a table that already has other customers. The CRP is *exchangeable* which is useful property in which the joint distribution of the table assignments z is independent of their order. That is, for any permutation of the integers $\ :$

$$p(z_1, z_2, \ldots) = p(z_{\ (1)}, z_{\ (2)}, \ldots). \tag{16}$$

3.2 MODEL

3.2.1 *Generative process for the theory* $p(T)$

The theory T represents what PWL believes to be true and is analogous to the internal mental model that humans create as they make observations of the world around them. More precisely, in PWM, the theory T is a collection of axioms $a_1, a_2, \ldots$, where each axiom a_i is a formula of higher-order logic. We choose a fairly simple prior for $p(T)$ for ease of implementation and rapid prototyping, but it is straightforward to

substitute $p(T)$ with a more complex prior. Specifically $a_1, a_2, \ldots$ are distributed according to a Dirichlet process.

$$G_a \quad \mathrm{DP}(H_a, \ ), \tag{17}$$

$$a_1, a_2, \ldots \quad G_a, \tag{18}$$

where H_a is the *base distribution* and $\ = 0.1$ is the concentration parameter.

The base distribution H_a is a distribution over logical forms in higher-order logic. To generate a sample from H_a, we sample each node in the expression tree of the logical form top-down, starting from the root. For each node in the expression tree:

1. Sample the operator at this node (i.e. atom, conjunction ^, disjunction _, negation :, quantification $\forall x$, etc) from a categorical distribution.

2. If we sampled an operator with a fixed number of operands (e.g. : has one operand, ! has two operands, etc), then recursively sample each operand. If a quantifier is sampled, set its variable to the next unused variable, and add it to the list of available variables. The list of available variables is the set of variables already declared by an ancestor of the current node in the expression tree of the logical form.

3. If we sampled an operator with a variable number of operands (e.g. ^, _), then first sample the number of operands from a geometric distribution. Next, sample each operand recursively.

4. If this node is selected to be an atom (e.g. book(c_1)), then its predicate is sampled from a non-parametric distribution of predicate symbols H_p. The atom's argument(s) are each sampled as follows: if n_v is the number of available variables, then sample a variable uniformly at random with probability $\frac{1}{n_v + 1}$; otherwise, with probability $\frac{1}{n_v + 1}$, sample a constant from a distribution of constant symbols H_c.

H_c is a uniform distribution over $\{c_1, \ldots, c_{100}\}$. H_p is the Chinese restaurant process with concentration parameter $\ = 1$:

$$z_1 = 1,$$

$$z_{i+1} = \begin{cases} k & \text{with probability } \frac{n_k}{\ + i}, \\ k^{\text{new}} & \text{with probability } \frac{\ }{\ + i}, \end{cases}$$

$$_i = p_{z_i},$$

were $p_1, p_2, \ldots$ is the set of available predicate symbols, and $_1, \ _2, \ldots$ are the samples from H_p.

In our semantic formalism, logical forms will contain atoms of the form arg1(a) = b or arg2(a) = b where a is a variable or constant and

b is either a variable, constant, number, or string (e.g. `arg1(X) = jason`). We will discuss the semantic formalism in greater detail in section 4.5. To accommodate these kinds of atoms, with some fixed probability, H_a will generate such an atom. Next, a is sampled by selecting a variable uniformly at random with probability $\frac{1}{n_V+1}$; otherwise, with probability $\frac{1}{n_V+1}$, sample a constant from H_c. The right-side of the equality b can either be a variable, constant, string, or number, and so PWM first selects its type from a categorical distribution. If the type is chosen to be a number, string, or variable, its value is sampled uniformly. If the type is chosen to be a constant, b is sampled from H_c.

Names of entities are treated specially in this prior: The number of names available to each entity is sampled independently and identically from a very light-tailed distribution: for entity c_i the number of names $n_N(c_i)$, #fs : `name(`c_i`) = sg` is distributed according to $p(n_N(c_i) = k) \propto 1/k^2$. This ensures that the number of names for each entity is small (usually 1).

Sets are also treated specially in this prior: One kind of axiom that can be generated is one that declares the size of a set, such as `size( X.planet(X)) = 8`, which denotes that the size of the set of planets is 8. Note that this is not a closed world assumption. The size of each set is an unobserved random variable, just like any other axiom in the theory. In the prior, the size of each set is distributed according to a geometric distribution with parameter 10^{-4}. Sets can have any arity $k > 0$, in which case their elements are k-tuples.

The above generative process provides a way to compute the prior probability of any theory. The parameters and code for computing the above prior is available at `github.com/asaparov/PWL`.

3.2.1.1 *Deterministic constraints on the theory*

PWM imposes a handful of hard constraints on the theory T. Most importantly, T is required to be *globally consistent*: There is no proof of a contradiction ? from the axioms a_i in T. While this is a conceptually simple requirement, it is computationally expensive (generally undecideable even in first-order logic). But PWL does not search over all possible proofs for a contradiction. Rather, PWL enforces this constraint by keeping track of the known sets in the theory. A set is *known* if its set size axiom is used in a proof, as in `size( X.planet(X)) = 8`, or if the set appears as a subset/superset in a universally-quantified axiom, such as in $\forall$`x(cat(X) ! mammal(X))` where the set `X.cat(X)` is a subset of `X.mammal(X)`. PWL keeps track of the size of each set as well as its provable elements. So for any known set, T will contain an axiom that declares the size of the set, even if that axiom is not explicitly used in any proof. For each set, the function `provable` (in algorithm 1) computes which elements are provably members of that set. If the number of provable members of a set is greater than its size, or if an element is both provably a member and not a member

Algorithm 1: Given a higher-order logic formula A, with free variables $x_1, \ldots, x_n$, this algorithm computes the maximal set of n-tuples $(v_{1,1}, \ldots, v_{1,n}), \ldots, (v_{N,1}, \ldots, v_{N,n})$ such that for each i, $A[x_1 \mapsto v_{i,1}, \ldots, x_n \mapsto v_{i,n}]$ (i.e. the formula A where each variable x_j is substituted with the value $v_{i,j}$) is provably true from the axioms in the theory. The elements of the tuples $v_{i,j}$ are restricted to be either constants, numbers, or strings. Note that this function does not exhaustively consider all proofs of A. This function uses the helper function `unify` which performs unification: given two input formulas A and B, `unify(A, B)` computes and $'$, where maps from variables in A to terms, $'$ maps from variables in B to terms, such that the application of to A is identical to the application of $'$ to B: (A) = $'$(B). In this algorithm (and its helper functions), `unify` only returns for brevity.

```
1  function provable(formula A)
2      let S be an empty set
3      for each axiom a_i in the theory T do
4          u = unify(A, a_i)
5          let S' be the set of all tuples (v_1, ..., v_k) such that v_i = u(x_i), for all i
6          set S = S ∪ S'
7      set S = S ∪ provable_by_theorem(A)
8      set S = S ∪ provable_by_exclusion(A)
9      if A is a conjunction B_1 ∧ ... ∧ B_N
10         for i = 1 to N do S_i = provable(B_i)
11         return S ∩ (S_1 ∖ ... ∖ S_N)
12     else if A is a disjunction B_1 ∨ ... ∨ B_N
13         for i = 1 to N do S_i = provable(B_i)
14         return S ∪ (S_1 ∪ ... ∪ S_N)
15     else if A is a negation ¬B
16         return S ∪ disprovable(B)
17     else if A is an implication B_1 → B_2
18         S_1 = disprovable(B_1)
19         S_2 = provable(B_2)
20         return S ∪ (S_1 ∪ S_2)
21     else if A is an existential quantification ∃x_{n+1}.f(x_1, ..., x_{n+1})
22         S' = provable(f(x_1, ..., x_{n+1}))
23         let S = {f(v_1, ..., v_n) : (v_1, ..., v_{n+1}) ∈ S'}
24         return S ∪ S
25     else if A is a universal quantification
           ∀x_{n+1} ... ∀x_{n+k}(f(x_1, ..., x_{n+k}) → g(x_1, ..., x_{n+k}))
26         for each known set {y_1 ... y_m.h(y_1, ..., y_m)} in T do
27             retrieve S' the provable elements of {y_1 ... y_m.h(y_1, ..., y_m)}
28             if |S'| ≠ the size of {y_1 ... y_m.h(y_1, ..., y_m)} continue
29             let u = unify(f(x_1, ..., x_{n+k}), h(y_1, ..., y_m))
30             if u is null continue
31             if u(x_i) is not a variable for some i ∈ {n + 1, ..., n + k} continue
32             let S be an empty set
33             for each (v_1, ..., v_m) ∈ S' do
34                 let (v'_1, ..., v'_{n+k}) where v'_i = v_k if u(x_i) = y_k, and v'_i = u(x_i) if u(x_i)
                       is a constant, number, or string, for all i
35                 set S = S ∪ {(v'_1, ..., v'_n)}
36             Q = provable(g(x_1, ..., x_{n+k}))
37             let Q = {f(v_1, ..., v_n) : (v_1, ..., v_{n+k}) ∈ S ∖ Q}
38             set S = S ∪ Q
```

Algorithm 1: (continued)

```
else if A is an equality B1 = B2
    if B1 and B2 are the same expression  return the set of all tuples
    if B1 or B2 has form size( )
        if B2 has form size( )  swap B1 and B2
        for each axiom a_i with form c = y1 ::: ym.h(y1, :::, ym) where c is a
          constant and y1 ::: ym.h(y1, :::, ym) is a known set do
            let n be the size of the set y1 ::: ym.h(y1, :::, ym)
            u = unify( , c)
            u^0 = unify(B2, n)
            if u^0 is null  continue
            if there is an x_i such that u(x_i) ≠ u^0(x_i)  continue
            let S^0 be the set of all tuples (v1, :::, vn) where v_i = u(x_i) or
              v_i = u^0(x_i) for all i
            set S = S ∩ S^0
    else if A has the form number(x_i)
        let S^0 be the set of all tuples where the i^th element is a number
        return S ∩ S^0
    else if A is >  return the set of all tuples
    else if A is ?  return ?
    return S
```

of a set, the theory is found to be inconsistent. Even though this may not find all possible contradictions in the theory, we find that it suffices to find the contradictions that arise in our experiments. But it is possible that for some other set of inputs, this consistency checking will fail to find a contradiction. Whenever an axiom is added to the theory T, PWL checks whether there are new provable members of any set, and updates the stored list of provable members accordingly. And whenever an axiom is removed, PWL checks whenever any objects are no longer provable members of a set. In our experiments, we find that consistency checking is much more costly when the theory is large. For example, on the question-answering examples of the FictionalGeoQA dataset that have more than 100 observed sentences, the reasoning module spends 68.8% of its time performing consistency checking. Relaxing this constraint would be valuable in future research, as it could save significant computation, perhaps instead by only considering the sets *relevant* to the current task rather than all sets in the theory, or deferring consistency checks altogether.

For axioms of the form ! , PWL also keeps track of whether the antecedent is provably true. It does so by using the `provable` function (in algorithm 1). Whenever an axiom is added to the theory T, PWL must check whether the antecedents of these axioms become provably true, since this would imply the consequent is now provably true (which can have further downstream consequences, such as newly provable elements of sets), and algorithm 1 and its helper functions only consider these axioms when their antecedents are true. If the antecedent is not known to be true, the truth of the consequent has

Algorithm 2: Helper functions used by algorithm 1. `provable_by_theorem` checks whether the formula A is provable from axioms of the form ! or $\forall x_1 ::: \forall x_k($!). `provable_by_exclusion` checks whether A would imply that the number of provable elements of any set is greater than its size, which would be a contradiction, thereby proving : A.

```
1  function provable_by_theorem(formula A)
2      let S be an empty set
3      for each known set y_1 ::: y_m.h(y_1,:::,y_m) in T do
4          retrieve S^0 the provable elements of y_1 ::: y_m.h(y_1,:::,y_m)
5          let h_1(y_1,:::,y_m) ^ ::: ^ h_k(y_1,:::,y_m) be the conjuncts of
             h(y_1,:::,y_m)
6          for i = 1,:::,k do
7              let u = unify(A, h_i(y_1,:::,y_m))
8              if u is null continue
9              for each (v_1,:::,v_m) 2 S^0 do
10                 let (v^0_1,:::,v^0_n) where v^0_j = v_k if u(x_j) = y_k, and v^0_j = u(x_j) if u(x_j) is
                     a constant, number, or string, for all j
11             set S = S [ (v^0_1,:::,v^0_n)
12     for each axiom in T with form ! where is provably true do
13         let _1 ^ ::: ^ _k be the conjuncts of
14         for i = 1,:::,k do
15             let u = unify(A, _i)
16             if u is null continue
17             let (v^0_1,:::,v^0_n) where v^0_j = u(x_j) for all j
18             set S = S [ f(v^0_1,:::,v^0_n)g
19     return S
20 function provable_by_exclusion(formula A)
       /* for tractability, we do not consider nested proofs by
          exclusion, and we only consider particular forms for A */
21     if this function has already been called higher in the stack return ?
22     if A is not of the form 9x.f(x) or f(x) where x is a variable return ?
23     let S be an empty set
24     for each known set y_1 ::: y_m.h(y_1,:::,y_m) in T do
25         retrieve S^0 the provable elements of y_1 ::: y_m.h(y_1,:::,y_m)
26         if jS^0j 6 the size of y_1 ::: y_m.h(y_1,:::,y_m) continue
27         if A has form : let N =
28         else let N = : A
29         for each u where u = unify(N, ), is a subformula of h(y_1,:::,y_m), such
             that if is an axiom in T, provable(h(y_1,:::,y_m)) returns a newly
             provable element: (v_1,:::,v_m) 2 S^0 do
               /* if A were true, the set y_1 ::: y_m.h(y_1,:::,y_m) would
                  have too many elements, which is a contradiction */
30             let (v^0_1,:::,v^0_n) where v^0_i = v_k, if u(x_i) = y_k, and v^0_i = u(x_i) if u(x_i) is a
                 constant, number, or string, for all i
31             set S = S [ f(v^0_1,:::,v^0_n)g
32     return S
```

Algorithm 3: Given a higher-order logic formula A, with free variables $x_1, \ldots, x_n$, this algorithm computes the maximal set of n-tuples $(v_{1,1}, \ldots, v_{1,n}), \ldots, (v_{N,1}, \ldots, v_{N,n})$ such that for each i, $A[x_1 \mapsto v_{i,1}, \ldots, x_n \mapsto v_{i,n}]$ (i.e. the formula A where each variable x_j is substituted with the value $v_{i,j}$) is provably *false* from the axioms in the theory. The elements of the tuples $v_{i,j}$ are restricted to be either constants, numbers, or strings. Note that this function does not exhaustively consider all proofs of $\neg A$.

---

function `disprovable`(*formula* A)
- let S be an empty set
- **for each** *axiom* a_i *in the theory* T **do**
  - u = `unify`($\neg A, a_i$)
  - let S' be the set of all tuples $(v_1, \ldots, v_k)$ such that $v_i = u(x_i)$, for all i
  - set $S = S \cup S'$
- set $S = S \cup$ `provable_by_theorem`($\neg A$)
- set $S = S \cup$ `provable_by_exclusion`($\neg A$)
- **if** A *is a conjunction* $B_1 \wedge \cdots \wedge B_N$
  - **for** $i = 1$ *to* N **do** S_i = `disprovable`(B_i)
  - **return** $S \cup (S_1 \cup \cdots \cup S_N)$
- **else if** A *is a disjunction* $B_1 \vee \cdots \vee B_N$
  - **for** $i = 1$ *to* N **do** S_i = `disprovable`(B_i)
  - **return** $S \cup (S_1 \cap \cdots \cap S_N)$
- **else if** A *is a negation* $\neg B$
  - **return** $S \cup$ *`provable`*(B)
- **else if** A *is an implication* $B_1 \rightarrow B_2$
  - S_1 = `provable`(B_1)
  - S_2 = `disprovable`(B_2)
  - **return** $S \cup (S_1 \cap S_2)$
- **else if** A *is an existential quantification* $\exists x_{n+1} \cdots \exists x_{n+k}.f(x_1, \ldots, x_{n+k})$
  - **return** $S \cup$ *`exists_disprovable`*(A)
- **else if** A *is a universal quantification* $\forall x_{n+1}.f(x_1, \ldots, x_{n+1})$
  - S' = `provable`($f(x_1, \ldots, x_{n+1})$)
  - let $S = \{(v_1, \ldots, v_n) : (v_1, \ldots, v_{n+1}) \in S'\}$
  - **return** $S \cup S$
- **else if** A *is an equality* $B_1 = B_2$
  - **if** B_1 *and* B_2 *are the same* **return** $\perp$
  - **if** B_1 *or* B_2 *has form* $c(\ )$ *where* c *is a constant*
    - **if** B_2 *has form* $c(\ )$ swap B_1 and B_2
    - **for each** *axiom* a_i *with form* $c(\ ) = n$ *where* n *is a constant, number, or string* **do**
      - u = `unify`(,)
      - u' = `unify`(B_2, n)
      - **if** u *is null or* u' *is empty* **continue**
      - let S' be the set of all tuples $(v_1, \ldots, v_n)$ where $v_i = u(x_i)$ for all i, and $v_i \neq u'(x_i)$ if u' is not null
      - set $S = S \cup S'$

Algorithm 3: (continued)

```
    if B1 or B2 is c where c is a variable or constant, or there is an axiom B1 = c or
      B2 = c where c is a constant
        /* without loss of generality, suppose B1 satisfies the
           above condition; otherwise swap B1 and B2 */
        if B1 is a constant c or B1 = q is an axiom with constant c
            if B2 is a constant c' or B2 = c' is an axiom with constant c' and c' ≠ c
                return the set of all tuples
            else if B2 is a variable x_j
                let S' be the set of all tuples (v_1, ..., v_n) such that v_j ≠ c
                set S = S ∩ S'
        else if B1 is a variable x_i
            if B2 is a constant c' or B2 = c' is an axiom with constant c'
                let S' be the set of all tuples (v_1, ..., v_n) such that v_i ≠ c'
                set S = S ∩ S'
            else if B2 is a variable x_j
                let S' be the set of all tuples (v_1, ..., v_n) such that v_i ≠ v_j
                set S = S ∩ S'
    else if A has the form number(x_i)
        let S' be the set of all tuples where the i^th element is not a number
        return S ∩ S'
    else if A is >  return ?
    else if A is ?  return the set of all tuples
    return S
```

no effect on the theory and so the `provable` function can ignore it. Similarly, whenever an axiom is removed, PWL checks whether the antecedents are no longer provably true.

We place a handful of other constraints on the theory T: The name of an entity must be a string (and not a number or a constant). All constants are distinct; that is, $\mathtt{c}_i \neq \mathtt{c}_j$ for all $i \neq j$. This helps to alleviate identifiability issues, as otherwise, there would be a much larger number of semantically redundant theories: for any theory, a logically-equivalent theory could be obtained by applying any permutation on the constants. No event can be an argument of itself (e.g. there is no constant $\mathtt{c}_i$ such that $\mathtt{arg1}(\mathtt{c}_i) = \mathtt{c}_i$ or $\mathtt{arg2}(\mathtt{c}_i) = \mathtt{c}_i$). If a theory T satisfies all constraints, we write "T valid."

The deterministic constraints on the theory do complicate computation of the prior, since the generative process for generating T is *conditioned* on T being valid:

$$p(\mathsf{T} \mid \mathsf{T} \text{ valid}) = p(\mathsf{T}) / p(\mathsf{T} \text{ valid}), \tag{19}$$

$$\text{where } p(\mathsf{T} \text{ valid}) = \sum_{\mathsf{T}' : \mathsf{T}' \text{ valid}} p(\mathsf{T}') \tag{20}$$

is the probability that the above generative process produces a valid theory, which is equal to the sum of the probabilities of all valid theories, which is intractable to compute. However, we show in section 3.3

Algorithm 4: Helper function used by algorithm 3 that returns the values of the free variables that make the given existentially-quantified formula provably false.

```
1  function exists_disprovable(formula ∃x_{n+1}:::∃x_{n+k}.f(x_1,:::,x_{n+k}))
2      let C_1 ^ ::: ^ C_N be the conjuncts of f(x_1,:::,x_{n+k})
3      let    be an empty substitution map, and let I be an empty set
4      for i = 1,:::,N do
5          if C_i has the form x_j = c where c is a constant, variable, or string, or C_i has the
             form x_j = q and there is an axiom q = c where c is a constant, variable, or
             string
6              add x_j ↦ c to the substitution map
7          else I.add(i)
8      let    be conjunction with conjuncts C_i where i ∈ I
9      let   ' be the result of applying the substitution map    to the formula
10     let M be an initially empty map
11     for each conjunct of   ' with form f(x_{n+j}) where   z.f(z) is a known set do
12         retrieve S' the provable elements of   z.f(z)
13         if the size of   z.f(z) is 0   return the set of all tuples
14         else if |S'| is not equal to the size of   z.f(z)   continue
15         M.put(x_{n+j},S')
16     if the number of entries in M is k
17         let S   be the set of all tuples
18         for f(v_{n+1},:::,v_{n+k}) : v_{n+i} ∈ M.get(x_{n+i}) for all i do
19             let     be the result of the substituting all x_{n+i} for v_{n+i} in   '
20             set S   = S   \ disprovable(   )
21         set S = S ∩ S
22     if there are conjuncts in   ' with the form
         x_{n+i} =   x_{n+k+1}::: x_{n+k'}g(x_1,:::,x_{n+k'}) and size(x_{n+i}) = c
23         if c is not an integer or a variable   return the set of all tuples
24         S' = provable(g(x_1,:::,x_{n+k'}))
25         let S   = f(v_1,:::,v_{n+k}) : (v_1,:::,v_{n+k'}) ∈ S'g
26         for each (v_1,:::,v_{n+k}) ∈ S   do
27             let I be the number of times (v_1,:::,v_{n+k}) appears in S'
28             if c is an integer and I > c
29                 set S = S ∩ f(v_1,:::,v_n)g
30             else if c is a variable x_i and I > v_i or v_i is not an integer
31                 set S = S ∩ f(v_1,:::,v_n)g
32         for each known set   y_1::: y_m.h(y_1,:::,y_m) in T do
33             u = unify(  x_{n+k+1}::: x_{n+k'}g(x_1,:::,x_{n+k'}),
                     y_1::: y_m.h(y_1,:::,y_m))
34             if u is null   continue
35             let I be the size of   y_1::: y_m.h(y_1,:::,y_m)
36             if c is an integer and I ∉ c
37                 let S' be the set of all tuples (v_1,:::,v_n) where v_i = u(x_i), for all i
38                 set S = S ∩ S'
39             else if c is a variable x_r
40                 let S' be the set of all tuples (v_1,:::,v_n) where v_i = u(x_i), for all i,
                     and v_r ∉ I
41                 set S = S ∩ S'
42     for each known set   y_1::: y_m.h(y_1,:::,y_m) in T do
43         if the size of   y_1::: y_m.h(y_1,:::,y_m) is not 0   continue
44         if h(y_1,:::,y_m) has form ∃y_{m+1}:::∃y_{m'}.h'(y_1,:::,y_{m'})
45             let    = h'(y_1,:::,y_{m'})
46         else let    = h(y_1,:::,y_m)
47         u = unify(  ',   )
48         let S' be the set of all tuples (v_1,:::,v_n), where for each i such that u(x_i)
             is a constant or string, v_i = u(x_i)
49         set S = S ∩ S'
50     return S
```

that for inference, it suffices to be able to efficiently compute the ratio of prior probabilities:

$$\frac{p(T_1 \mid T_1 \text{ valid})}{p(T_2 \mid T_2 \text{ valid})} = \frac{p(T_1)p(T_2 \text{ valid})}{p(T_2)p(T_1 \text{ valid})} = \frac{p(T_1)}{p(T_2)}. \tag{21}$$

Additionally note that since the above constraints do not depend on the *order* of the axioms, constants, etc. (i.e. the constraints themselves are exchangeable), the distribution of T conditioned on T being valid is exchangeable.

Note that the `provable` function and its helper functions make heavy use of the `unify` function, which when given two input formulas A and B, finds substitutions and ^{0}, where is a map from the variables of A to higher-order terms, ^{0} is a map from the variables of B to higher-order terms, and applied to A is identical to ^{0} applied to B. In the pseudocode for `provable` and its helper functions shown in this thesis, `unify` function returns if such a map exists; otherwise, it returns *null*. Note that this is not the same as full higher-order unification, where the substituted formulas are equivalent under , , and reductions. For efficiency, `unify` only considers substitution maps from variables to variables, constants, numbers, or strings.

The `provable` function and its helper functions work with sets of tuples (possibly infinite), computing their unions and intersections. To do so efficiently, they make use of a sparse data structure to represent these sets, shown below:

```
1  class tuple_element
       /* supertype that represents a
          tuple element */
2  class tuple_constant
     extends tuple_element
3      int constant_id
4  class tuple_string
     extends tuple_element
5      string str
6  class tuple_number
     extends tuple_element
7      number num
8  class interval
9      number min
10     number max
11     bool is_min_inclusive
12     bool is_max_inclusive
13 class tuple_any_number
     extends tuple_element
14     array<interval> intervals
15 class tuple_any extends tuple_element
       /* represents the set of all
          values except those in
          'excluded'  */
16     array<tuple_element> excluded
17 class tuple_set
18     array<tuple_element> elements
19     array<pair<int,int>> equal
20     array<pair<int,int>> unequal
21     array<pair<int,int>> ge
```

In the `tuple_set` data structure, the `elements` array represents the elements of the tuple: so for a set of tuples S, `elements[i]` represents $f v_i : (v_1, \ldots, v_n)$ *2* Sg. the `equal` field represents the *equality constraints* on the set of tuples: it contains pairs of indices (i, j) such that for any tuple $(v_1, \ldots, v_n)$ in the set S, $v_i = v_j$. Similarly, the `unequal` field represents the *inequality constraints*: it contains pairs of indices (i, j) such that for any tuple $(v_1, \ldots, v_n)$ in the set S, v_i ~~6=~~ v_j. Finally, the

ge field contains *greater-than-or-equal-to constraints*: it contains pairs of indices (i, j) such that for any tuple $(v_1, \ldots, v_n)$ in the set S, $v_i > v_j$.

Equipped with this data structure, the sets of tuples in the `provable` function and its helper functions can be represented as lists of disjoint `tuple_set` objects, where the list represents the union of the corresponding sets.

CHECKING THE CONSISTENCY OF SET SIZES: Many of the axioms in the theory will be of the form `size( x.f(x)) = n` where `n > 0` is an integer. These axioms declare that the size of the set $\{x : f(x)\}$ is n (the set of all objects x such that f(x) is true). Consider the axioms: `size( x.cat(x)) = 3`, `size( x.small(x)) = 2`, `size( x(cat(x) ^ small(x))) = 0`, and `size( x(cat(x) _ small(x))) = 4`. These axioms state that the number of cats is 3, the number of small objects is 2, the number of small cats is 0, and the number of objects that are either cats or small is 4. But this is impossible since for any two sets A and B, $|A \cup B| = |A| + |B| - |A \setminus B|$. And so in the above example, the size of the set `x(cat(x) _ small(x))` must be at least 5. It is possible to add these axioms to the theory and rely on the above consistency checking mechanisms (the `provable` function) in order to check for the consistency of set sizes. However, this would be fairly inefficient.

Instead, PWL uses a specialized data structure to maintain the consistency of set sizes. This data structure consists of a directed graph G, where each vertex in G corresponds to a known set. Each directed edge corresponds to the superset relation: if A is a superset of B, there is a directed path from the vertex u to the vertex v in G, where u corresponds to A and v corresponds to B. We take care to avoid adding superfluous edges: if there is an edge (u, v) in G and there is an edge (v, w) in G, then there is no edge (u, w). This serves to keep G as sparse as possible. This graph structure enables efficient retrieval of the provable elements of any known set: if e is a provable element of the set A, then e is a provable element of every superset of A (i.e. every ancestor of the vertex corresponding to A in G). The graph structure also helps to determine whether two sets A and B are disjoint: check whether any ancestor of the vertex corresponding to $A \setminus B$ has size 0. Perhaps most importantly, the graph structure helps to determine, for each set A, the minimum size of A that is consistent with the sizes of the other known sets. The following constraint must hold:

$$|A| > \max_{\substack{(B_1, \ldots, B_n) : B_i \subseteq A, \\ B_i \setminus B_j = \varnothing \text{ for all } i \neq j}} \sum_{i=1}^{n} |B_i|. \tag{22}$$

That is, for any sets $B_1, \ldots, B_n$ which are disjoint subsets of A (n could be 1), the size of A must be at least the sum of the sizes of all B_i. This constraint also induces an upper bound on the size of A, since A itself may be a subset of other sets. To find the collection of subsets $B_1, \ldots, B_n$ that maximizes their sum, we consider the "disjointedness"

Algorithm 5: Modified Bron-Kerbosch algorithm to find the disjoint clique of vertices $c_1, \ldots, c_k$ that maximizes $\sum_{i=1}^{k} w(c_i)$, where $w(c_i)$ is the weight of the vertex c_i, each c_i is a descendant of the given input vertex v, and for all $i \neq j$, the set corresponding to the vertex c_i is disjoint with the set corresponding to c_j.

```
1  function search_helper(G, vertex list M, vertex list X, vertex u, vertex v)
2      if u and v are disjoint in G
3          for each n ∈ M is an ancestor of v do M.remove(n)
4          M.add(v)
5      else
6          for each c child vertex of v do
7              if w(c) = 0 or ∃n ∈ M ∩ X such that n is an ancestor of c continue
8              search_helper(G, M, X, u, c)
9  function max_weight_disjoint_clique(graph G, vertex r)
10     let Q be an empty priority queue
11     let {u_1, ..., u_n} = {u : w(u) ≠ 0, (r, u) is an edge in G} be the immediate
         subsets (children) of r with non-zero weight
12     for i = 1, ..., n do
13         let N = {u_j : j < i ∧ u_i and u_j are disjoint}
14         let   = w(u_i) + Σ_{u∈N} w(u)
15         Q.add( new search state (?, N, ?, u_i) with priority   )
16     let L be an initially empty list of completed cliques
17     while Q is not empty do
18         (C, N, X, v) = Q.pop() with priority
19         if   ≤ highest weight of a clique in L break
20         let M be an initially empty list
21         for each x ∈ X do search_helper(M, M, v, x)
22         copy X' from M
23         for each n ∈ N do search_helper(M, X', v, n)
24         let {u_1, ..., u_n} be M ∩ X'
25         for i = 1, ..., n do
26             let N   = {u_j : j < i}
27             let X   = M ∩ (N ∩ {u_i})
28             let   ' = w(v) + Σ_{c∈C} w(c) + Σ_{j=i}^{n} w(u_i)
29             Q.push( new search state (C ∪ {v}, N  , X  , u_i) with priority   ')
30         if both M and X are empty L.add(C ∪ {v})
31         let {u_1, ..., u_n} = {u : w(u) ≠ 0, (v, u) is an edge in G} be the immediate
             subsets (children) of v with non-zero weight
32         for i = 1, ..., n do
33             let N   = {u_j : j < i ∧ u_i and u_j are disjoint}
34             let   ' = w(u_i) + Σ_{c∈C} w(c) + Σ_{u∈N} w(u)
35             Q.push( new search state (C, N  , M, u_i) with priority   ')
36     return maximum weight clique in L
```

graph of G: Let D be a graph with the same vertices as G, and there is an edge (u, v) in D if and only if the sets corresponding to u and v in G are disjoint. Thus, D is undirected, unlike G. Each vertex in D is weighted according to the size of the corresponding set. The problem of finding the maximal disjoint subsets $B_1, \ldots, B_n$ is reduced to the problem of finding the maximal clique in D, consisting only of the descendant vertices of A, that maximizes the sum of the weights. To perform this optimization, PWL uses a modified version of the Bron-Kerbosch algorithm (Bron and Kerbosch, 1973), shown in algorithm 5.

The algorithm is an application of *branch-and-bound*: it starts by considering the set of all cliques of vertices in D which are descendants of the input vertex r in G. Next, on line 12, it subdivides this set to a collection of disjoint subsets (the "branch" step), where each subset is the set of cliques that contains either v or a descendant of v, where v is a child vertex of r. Each subset is pushed into the priority queue Q. Then algorithm repeats this process: each element in the priority queue Q is a tuple (C, N, X, v) (i.e. a search state). This tuple represents a set of candidate cliques S, where each clique contains all the vertices in C, some of the vertices in N, and none of the vertices in X, and it contains v or least one of the descendants of v. All vertices $v \in N \cup X$ are disjoint with each vertex $c \in C \cup \{v\}$. At each iteration of the main loop (on line 17), the algorithm subdivides this set S (i.e. creates new search states) by considering moving each vertex from N into the clique $C \cup \{v\}$. If v has child vertices, the algorithm also creates new search states by considering replacing v with one of its children. As such, these new search states are disjoint. The priority of each search state S is an upper bound on the weight of any clique in the set of candidate cliques represented by that state: $h(S) = w(v) + \sum_{c \in C} w(C) + \sum_{n \in N} w(n)$, where $w(x)$ is the weight of the vertex x. Note that $h(S)$ is a valid upper bound on the weight of any clique in the set S, since the largest possible clique in S is one that includes all of the vertices in $C \cup N \cup \{v\}$. When the algorithm processes a search state where there are no further candidate vertices N that can be added to the clique, the clique is maximal and is added to a list of completed cliques L. Once the highest priority in the priority queue is less than or equal to the highest weight of a completed clique in L, the best clique in L is guaranteed to be optimal.

Though the maximum weighted clique problem is NP-hard and algorithm 5 has exponential worst-case complexity, it is more efficient when the graph is sparse. In our experiments, the graphs were small (at most 120 vertices) and sparse enough that this algorithm would always terminate quickly. However, it remains to be seen how it will fare when the theory is much larger, containing many more known sets. And future research to relax this constraint would be valuable, perhaps by restricting the optimization to local regions of the graph.

3.2.1.2 *Properties of the theory prior* p(T)

We emphasize that the distribution for p(T) was chosen for simplicity and ease of implementation, and it worked well enough in our experiments. However, there is likely a large family of distributions that would work similarly well. Nevertheless, this prior does exhibit useful properties for a domain- and task-general model of reasoning:

- *Occam's razor:* Smaller/simpler theories are given higher probability than larger and more complex theories, both in terms of the number of axioms but also in the complexity of each axiom.
- *Consistency:* Inconsistent theories are discouraged or impossible.
- Entities tend to have a unique name. Our prior above encodes one direction of this prior belief: each entity is unlikely to have many names. However, the prior does not discourage one name from referring to multiple entities.
- Entities tend to have a unique type. Note however that this does not discourage types provable by subsumption. For example, if the theory has the axioms $\texttt{novel}(\texttt{c}_1)$ and $\forall\texttt{x}(\texttt{novel}(\texttt{x}) \rightarrow \texttt{book}(\texttt{x}))$, even though $\texttt{book}(\texttt{c}_1)$ is provable, it is not an axiom in this example and the prior only applies to axioms.

3.2.2 *Generative process for the proofs* p($_i$ | T)

PWM uses *natural deduction*, a well-studied proof calculus, for its proofs (Gentzen, 1935, 1969). Pfenning (2004) provides an accessible introduction. Figure 9 illustrates a simple example of a natural deduction proof. Each horizontal line is a proof step, with the (zero or more) formulas above the line being the *premises* of that proof step, and the single formula below the line being the conclusion of that proof step. Each proof step has a label to the right of the line. For example, the "^I" step denotes *conjunction introduction*: given that A and B are true, this step concludes that A ^ B is true, where A and B can be any formula. A natural deduction proof can use axioms in its proof steps (the axioms are given by proof steps labeled "Ax"). In the example in figure 9, the proof contains the axiom A ^ B. Natural deduction is *complete* in that if any higher-order formula is true (under Henkin semantics), there is a natural deduction proof of (Henkin, 1950), which is a very useful property for generality.

We can write any natural deduction proof $_i$ as a sequence of proof steps $_i$, ($_{i,1}, \ldots,$ $_{i,k}$) by traversing the proof tree in prefix order. We define a simple generative process for $_i$:

1. First sample the length of the proof k from a Poisson distribution with parameter 20.

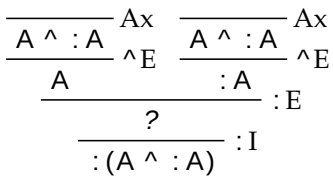


Figure 9: An example of a proof of $\neg(A \wedge \neg A)$. The proof starts with the axiom $A \wedge \neg A$. By conjunction elimination ($\wedge$E), we conclude from this axiom that both A and $\neg A$ are true. By negation elimination ($\neg$E), we conclude from the fact that both A and $\neg A$ are true that there is a contradiction $\bot$. Finally, from the contradiction, via negation introduction ($\neg$I), we conclude that the negation of the original axiom is true: $\neg(A \wedge \neg A)$. The tree structure of natural deduction proofs is visible in this example, where the two leaves are the axioms at the top and the root is the conclusion at the bottom.

2. For each $j = 1, \ldots, k$: Select a deduction rule from the proof calculus with a categorical distribution. If the Ax rule is selected, then simply take the next available axiom from the theory $T = a_1, a_2, \ldots$ If the deduction rule requires premises, then each premise is selected uniformly at random from $i_{,1}, \ldots, i_{,j-1}$. Some deduction rules will require additional parameters:

   a) If the selected rule is conjunction elimination $\wedge$E, its premise is a conjunction $\_1 \wedge \ldots \wedge \_n$ and its conclusion is $\_{i_1} \wedge \ldots \wedge \_{i_k}$ where $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ and $i_1 < \ldots < i_m$. We sample m from a Poisson distribution with parameter 1.5, and each increment $i_{j+1} - i_j$ is sampled from a Poisson distribution with parameter 2 (the initial index i_1 is also sampled from a Poisson distribution with parameter 2).

   b) If the selected rule is disjunction introduction $\vee$I, its premise is a formula $\_$ and its conclusion is $\_ \vee \_$. We select $\_$ uniformly at random from $\_{i,1}, \ldots, \_{i,j-1}$, as with all other deduction rules, but we also need to generate $\_$. For simplicity, we choose $\_$ uniformly at random from the set of all possible logical forms with depth less than N where N is large. While this is very unrealistic, it is simple to implement. This approach sufficed in our experiments since they did not often produce proofs that contained this deduction rule. A better approach could be to sample $\_$ from H_a.

   c) If the selected rule is universal introduction $\forall$I, its premise is a formula $\_$ and its conclusion is $\forall x.\ [a \to x]\_$ where $[a \to x]\_$ is the substitution of the parameter a with the variable x in the formula $\_$. As with all other deduction rules, the premise $\_$ is selected uniformly at random from $\_{i,1}, \ldots, \_{i,j-1}$. The parameter a is sampled uniformly from the set of parameters that appear in $\_$.

d) If the selected rule is universal elimination ∀E, its premise is a formula ∀x. and its conclusion is [x ↦ c] for some term c. As with all other deduction rules, the premise ∀x. is selected uniformly at random from $_{i,1}, \ldots, _{i,j-1}$. The term c is drawn from a Chinese restaurant process with concentration parameter = 1:

$$z_1 = 1,$$
$$z_{i+1} = \begin{cases} k & \text{with probability } \frac{n_k}{+i}, \\ k^{\text{new}} & \text{with probability } \frac{}{+i}, \end{cases}$$
$$_i = t_{z_i},$$

where $t_1, t_2, \ldots$ is a list of available terms, and $_1, _2, \ldots$ are the samples from the CRP (c being among them).

e) If the selected rule is existential introduction ∃I, its premise is a formula [x ↦ c] and its conclusion is ∃x. , where x is a variable and c is a term. As with all other deduction rules, the premise formula is selected uniformly at random from $_{i,1}, \ldots, _{i,j-1}$. Note that it is not necessary to replace every occurrence of c in [x ↦ c] with x. For example, from see(kate, kate), we can conclude ∃x.see(x, kate). To select which occurrences of c to replace with the variable x, we need to generate a list of indices $i_1, \ldots, i_m$, where each index identifies a node in the prefix ordering of nodes of the expression tree of [x ↦ c]. For example, in see(kate, kate), the subexpression with index 1 is see(kate, kate). The subexpression with index 2 is see. The subexpression with index 3 is the first occurrence of kate, and so on. So in order to conclude ∃x.see(x, kate) from see(kate, kate), the list of indices would be {3}. To generate the list of indices $i_1, \ldots, i_m$ where $i_1 < \cdots < i_m$, we first sample m from a Poisson distribution with parameter 1.5. Next, we sample each increment $i_{j+1} - i_j$ is sampled from a Poisson distribution with parameter 4 (the initial index i_1 is also sampled from a Poisson distribution with parameter 4).

f) If the selected rule is equality elimination = E, its premises are a formula [p ↦ X] and an equality X = Y. Its conclusion is [p ↦ Y], which is identical to the premise except some of its occurrence of X have been replaced with Y. But just as with ∃I above, it is not necessary to replace every occurrence of X with Y. For example, from see(kate, kate) and kate = sister(matt), we can conclude see(kate, sister(matt)). So again we need a list of indices $i_1, \ldots, i_m$ where $i_1 < \cdots < i_m$ to indicate which occurrences of X to replace with Y. We generate this list from the same distribution as the

indices for the ∃I rule, as described above: First sample m from a Poisson distribution with parameter 1.5. Next, we sample each increment $i_{j+1} - i_j$ is sampled from a Poisson distribution with parameter 4 (the initial index i_1 is also sampled from a Poisson distribution with parameter 4).

The above generative process may produce invalid proofs: For example, it may produce a forest rather than a single proof tree, or some of the deduction steps may have premises with types that do not match the expected type for that deduction step (e.g. the premise of conjunction elimination ∧E is required to be a conjunction, but the above process may produce proofs where the premise is not always a conjunction). Thus, π_i is sampled conditioned on π_i being a *valid* proof. Just as with $p(T)$ in equation 19, this conditioning causes $p(\pi_i \mid T)$ to be intractable to compute. However, only the ratio of the prior probability is needed for inference, which can be computed efficiently:

$$\frac{p(\pi_i \mid T, \pi_i \text{ valid})}{p(\pi'_i \mid T, \pi'_i \text{ valid})} = \frac{p(\pi_i \mid T)p(\pi'_i \text{ valid} \mid T)}{p(\pi'_i \mid T)p(\pi_i \text{ valid} \mid T)} = \frac{p(\pi_i \mid T)}{p(\pi'_i \mid T)}. \tag{23}$$

PWL was initially implemented assuming classical logic, since we believed that human reasoning aligns most commonly with classical logic. However, it is easy to adapt PWL to use other logics, such as *intuitionistic logic*. Intuitionistic logic is identical to classical logic except that the *law of the excluded middle* $A \vee \neg A$ is not a theorem (see figure 28 for an example in the ProofWriter dataset where the two logics disagree). The interpretable nature of the reasoning module makes it easy to adapt PWL to other kinds of logic or proof calculi. One of our experiments uses the ProofWriter dataset, which was constructed with intuitionistic logic. In order to measure the performance of PWL on the ProofWriter dataset, PWL supports reasoning with both classical and intuitionistic logic. A flag allows the user to switch between the two.

We emphasize that all of the parameters in the above prior are fixed, and so PWL does not learn them from the data. This worked well enough in our experiments, but a richer prior may be required for larger theories, where some of the parameters are random variables and are themselves learned.

3.3 INFERENCE AND IMPLEMENTATION

Having described the generative process for the theory T and proofs $\pi, \{\pi_1, \ldots, \pi_n\}$, we now describe inference. Given logical forms $x, \{x_1, \ldots, x_n\}$, the goal is to compute the posterior distribution of T and π such that the conclusion of the each proof π_i is x_i. That is, PWL performs abduction: recovering the latent theory and proofs that explain/entail the given observed logical forms. Real natural language is not unambiguous, and real-world observations often have

Algorithm 6: Pseudocode for proof initialization. If any new axiom violates the deterministic constraints in section 3.2.1.1, the function returns *null*.

```
1  function init_proof(formula A)
2    if A is a conjunction B_1 ^ ::: ^ B_n
3      for i = 1 to n do  _i = init_proof(B_i)
4      return  (_1 ::: _n) / B_1 ^ ::: ^ B_n  ^I
5    else if A is a disjunction B_1 _ ::: _ B_n
6      l = shuffle(1,:::,n)
7      for i 2 l do
8        _i = init_proof(B_i)
9        if _i ≠ null return  _i / B_1 _ ::: _ B_n  _I
10   else if A is a negation :B
11     return init_disproof(B)
12   else if A is an implication B_1 ! B_2
13     if using classical logic
14       l = shuffle(1,2)
15       for i 2 l do
16         if i = 1
17           _1 = init_disproof(B_1)
18           if _1 ≠ null return  [B_1]^Ax / ? / B_2 ?E / B_1 ! B_2 !I  :E
19         else
20           _2 = init_proof(B_2)
21           if _2 ≠ null return  _2 / B_1 ! B_2  ! I
22     else if using intuitionistic logic
23       return  B_1 ! B_2  Ax
24   else if A is an existential quantification 9x.f(x)
25     let C be the set of known constants, numbers, and strings in T, and the
         new constant c
26     l = swap(shuffle(C))
27     for c 2 l do
28       _c = init_proof(f(c))
29       if _c ≠ null return  c / 9x.f(x)  9I
30   else if A is a universal quantification 8x.f(x)
31     return  8x.f(x)  Ax
32   else if A is an equality B_1 = B_2
33     return  B_1 = B_2  Ax
34   else if A is a set membership statement s(c) where s was defined s =  x.f(x)
35       = init_proof(f(c))
36     return  [s = x.f(x)]^Ax / s(c)  =E
37   else if A is an atom (e.g. book(great_gatsby))
38     return  A  Ax
39   else return null
```

Algorithm 7: Helper function for `init_proof` (shown in algorithm 6) that returns a proof that the given formula A is *false*. If any new axiom violates the deterministic constraints in section 3.2.1.1, the function returns *null*.

```
function init_disproof(formula A)
  if A is a conjunction B1 ^ ::: ^ Bn
    I = shuffle(1,:::,n)
    for i 2 I do
      i = init_disproof(Bi)
                                  ―――――――――――― Ax
                                  B1 ^ ::: ^ Bn
                                  ―――――――――――― ^E
      if  i 6= null  return    i        Bi
                           ――――――――――――――― :E
                                  ?
                           ――――――――――――――― :I
                           :(B1 ^ ::: ^ Bn)
  else if A is a disjunction B1 _ ::: _ Bn
    for i = 1 to n do  i = init_disproof(Bi)
                                         ――― Ax               ――― Ax
                                     1   B1                n  Bn
                  ―――――――――――― Ax   ――――――――― :E   :::   ――――――――― :E
    return        B1 _ ::: _ Bn          ?                    ?
                  ―――――――――――――――――――――――――――――――――――――――――――――――― _E
                                         ?
                                   ――――――――――――――― :I
                                   :(B1 _ ::: _ Bn)
  else if A is a negation :B
    return init_proof(B)
  else if A is an implication B1 ! B2
     1 = init_proof(B1)
     2 = init_proof(B2)
                      ―――――――― Ax
                   1  B1 ! B2
                  ―――――――――――― ! E
    return     2      B2
             ――――――――――――――― :E
                      ?
             ――――――――――――――― :I
               :(B1 ! B2)
  else if A is an existential quantification 9x.f(x)
             ――――――――――――――― Ax   ――――――――――――――――――――――――― Ax
    return   size( x.f(x)) = 0     8S(size(S) = 0 $ :9x.S(x))
             ―――――――――――――――――――――――――――――――――――――――――――――――― =E
                              :9x.f(x)
  else if A is a universal quantification 8x.f(x)
    let C be the set of known constants, numbers, and strings in T, and the
      new constant c
    I = swap(shuffle(C))
    for c 2 I do
      c = init_disproof(f(c))
                                   ―――――――― Ax
                                   8x.f(x)
                                   ―――――――― 8E
      if  c 6= null  return    c     f(c)
                           ――――――――――――――― :E
                                  ?
                              ――――――――― :I
                              :8x.f(x)
  else if A is an equality B1 = B2
             ――――――――― Ax
    return   B1 6= B2
  else if A is a set membership statement s(c) where s was defined s =  x.f(x)
     = init_disproof(f(c))
                ――――――――――― Ax
    return      s =  x.f(x)
             ―――――――――――――――― =E
                   :s(c)
  else if A is an atom (e.g. book(great_gatsby))
             ――― Ax
    return   :A
  else return null
```

multiple probable explanations. In addition, there exist sentences that expresses information about this uncertainty, such as "A liquid water ocean probably exists under the surface of Enceladus." These sentences must have been generated with explicit awareness of this uncertainty. Therefore, rather than inferring a single most probable theory, PWL endeavors to infer the posterior distribution of the theory given the observations.

To this end, PWL uses Metropolis-Hastings (MH). PWL performs inference in a streaming fashion, starting by considering only the first sentence (i.e. the case $n = 1$) to obtain MH samples from $p(\ _1, T \mid x_1)$. Then, for every new logical form x_n, PWL uses the last sample from $p(\ _1, \ldots, \ _{n-1}, T \mid x_1, \ldots, x_{n-1})$ as a starting point of the Markov chain and then obtains MH samples from $p(\ _1, \ldots, \ _n, T \mid x_1, \ldots, x_n)$. This warm-start initialization serves to dramatically reduce the number of iterations needed to mix the Markov chain. To obtain the MH samples, the proof of each new logical form $\ _n^{(0)}$ is initialized using algorithm 6, whereas the proofs of previous logical forms are kept from the last MH sample. The axioms in these proofs constitute the theory sample $T^{(0)}$. Then, for each iteration $t = 1, \ldots, N_{\text{iter}}$, MH proposes a mutation to one or more proofs in $\ ^{(t)}$. The possible mutations are listed in table 1. These mutations may change axioms in $T^{(t)}$. Let $T^0, \ _i^0$ be the newly proposed theory and proofs. Then, compute the acceptance probability:

$$\min\left(1, \frac{p(T^0)}{p(T^{(t)})} \prod_{i=1}^{\infty} \frac{p(\ _i^0 \mid T^0)}{p(\ _i^{(t)} \mid T^{(t)})} \frac{g(T^{(t)}, \ _i^{(t)} \mid T^0, \ _i^0)}{g(T^0, \ _i^0 \mid T^{(t)}, \ _i^{(t)})}\right), \tag{24}$$

where $g(T^0, \ _i^0 \mid T^{(t)}, \ _i^{(t)})$ is the probability of proposing the mutation from $T^{(t)}, \ _i^{(t)}$ to $T^0, \ _i^0$, and $g(T^{(t)}, \ _i^{(t)} \mid T^0, \ _i^0)$ is the probability of the *inverse* of this mutation. Since this quantity depends only on the *ratio* of probabilities, it can be computed efficiently (see equations 21 and 23). Once this quantity is computed, sample from a Bernoulli with this quantity as its parameter. If it succeeds, MH accepts the proposed theory and proofs as the next sample: $T^{(t+1)} = T^0$ and $\ _i^{(t+1)} = \ _i^0$. Otherwise, reject the proposal and keep the old sample: $T^{(t+1)} = T^{(t)}$ and $\ _i^{(t+1)} = \ _i^{(t)}$. If every possible theory and proof is reachable from the initial theory by a sequence of mutations, then with sufficiently many iterations, the samples $T^{(t)}$ and $\ _i^{(t)}$ will be distributed according to the true posterior $p(T, \ _1, \ldots, \ _n \mid x_1, \ldots, x_n)$. In our experiments, we use $N_{\text{iter}} = 400$ or $N_{\text{iter}} = 600$ iterations of MH, which we find provides good estimates of the posterior probabilities. If only a subset of possible theories and proofs are reachable from the initial theory, the MH samples will be distributed according to the true posterior *conditioned* on that subset. This may be good enough for many applications, particularly if the theories in the subset have desirable properties such as superior tractability. However, the subset cannot be made too small as then PWL would lose generality.

Proposal	Probability of selecting proposal
Select a grounded atomic axiom (e.g. square(c_1)) and propose to replace it with an instantiation of a universal quantification (e.g. $\forall x$(rectangle(x) ^ rhombus(x) ! square(x))), where the antecedent conjuncts are selected uniformly at random from the other grounded atomic axioms for the constant c_1: rectangle(c_1), rhombus(c_1), etc.	$\frac{1}{N}$
The inverse of the above proposal: select an instantiation of a universal quantification and replace it with a grounded atomic axiom.	$\frac{1}{N}$
Select an axiom that declares the size of a set (e.g. of the form size(x.state(x)) = 50), and propose to change the size of the set by sampling from the prior distribution, conditioned on the maximum and minimum allowable set size (to maintain consistency).	$\frac{1}{N}$
Select a node from a proof tree in $_1, \dots, _n$ of type _I, ! I, or $\exists$I (and also disproofs of conjunctions, if using classical logic). These nodes were created in algorithm 6 on lines 6, 14, and 26, respectively, where for each node, a single premise was selected out of a number of possible premises. This proposal naturally follows from the desire to explore other selections by re-sampling the proof: it simply calls `init_proof` again on the formula at this proof node. However, it is difficult to compute the probability of this proposal if `init_proof` is used as written. Instead, we replace the function calls to `shuffle(...)` or `swap(shuffle(...))` with `first(shuffle(...))` or `first(swap(shuffle(...)))`, where `first` simply returns the first element from its input list. This makes `init_proof` much cheaper to compute, but much more likely to fail, since it only considers one possible proof for the given formula rather than searching over a large space of possible proofs. In case of failure, this proposal simply tries `init_proof` again, and repeats this process until it succeeds. In our experiments, we find that this modified `init_proof` function fails roughly 70.8% of the time.	$\frac{1}{N}$
Merge: Select a "mergeable" event; that is, three constants (c_i, c_j, c_k) such that arg1(c_i) = c_j, arg2(c_i) = c_k, and t(c_i) for some constant t are axioms, and there also exist constants ($c_{i'}, c_{j'}, c_{k'}$) such that $i' > i$, arg1($c_{i'}$) = $c_{j'}$, arg2($c_{i'}$) = $c_{k'}$, and t($c_{i'}$) are axioms. Next, propose to merge $c_{i'}$ with c_i by replacing all instances of $c_{i'}$ with c_i in the proof trees, $c_{j'}$ with c_j, and $c_{k'}$ with c_k. This proposal is not necessary in that these changes are reachable by a sequence of other proposals, but those proposals may have low probability, and so this proposal serves to more easily escape local maxima.	$\frac{}{N}$
Split: The inverse of the above proposal.	$\frac{}{N}$

Table 1: A list of the Metropolis-Hastings proposals implemented in PWL thus far. N, here, is a normalization term: N = jAj + jUj + jCj + jPj + jMj + jSj where: A is the set of grounded atomic axioms in T (e.g. square(c_1)), U is the set of universally-quantified axioms that can be eliminated by the second proposal, C is the set of axioms that declare the size of a set (e.g. size(A) = 4), P is the set of nodes of type _I, ! I, or $\exists$I (and also disproofs of conjunctions, if using classical logic) in the proofs $_1, \dots, _n$, M is the set of "mergeable" events (described above), and S is the set of "splittable" events. In our experiments, = 2 and = 0.001.

The function `init_proof` in algorithm 6 recursively calls `init_disproof`, shown in algorithm 7, which closely mirrors the structure of `init_proof`. The purpose of `init_proof` is to find *some* proof of a given higher-order logic formula, or return *null* if none exists. Its task is abduction, which is computationally easier than theorem proving, since new axioms can be created as needed. The returned proof need not be "optimal" since it serves as the initial state for MH, which will further refine the proof. The validity of the proofs is guaranteed by the fact that `init_proof` only returns valid proofs and MH only proposes mutations to proofs that preserve the correctness of the proof.

In algorithm 6, the `shuffle` function uniformly shuffles its input. The `swap` function (called on line 26, in the case of existential quantification) will randomly select an element in its input list to swap with the first element. The probability of moving an element c to the front of the list is computed as follows: Recursively inspect the atoms in the formula $f(c)$ and count the number of "matching" atoms: The atoms $t(c)$ or $c(t)$ is considered "matching" if it is provable in T. Next, count the number of "mismatching" axioms: for each atom $t(c)$ in the formula $f(c)$, an axiom $t'(c)$ is "mismatching" if $t \not\leftrightarrow t'$. And similarly for each atom $c(t)$ in the formula $f(c)$, an axiom $c(t')$ is "mismatching" if $t \not\leftrightarrow t'$. Let n be the number of "matching" atoms and m be the number of "mismatching" axioms, then the probability of moving c to the front of the list is proportional to $\exp(n - 2m)$. This greatly increases the chance of finding a high-probability proof in the first iteration of the loop on line 27, and since this function is also used in an MH proposal, it dramatically improves the acceptance rate. This reduces the number of MH iterations needed to sufficiently mix the Markov chain.

Note that it is possible for `init_proof` to fail even if the given logical form is not inconsistent with the other observations. Consider the case where the observed logical forms are: (1) `planet(pluto) _ dwarf_planet(pluto)`, and (2) `: planet(pluto)`. Suppose the first logical form is added to the theory, and the last sample of T in the Markov chain has the axiom `planet(pluto)`, then when PWL adds the second logical form, `: planet(pluto)`, `init_proof` will fail to find a valid proof. When this happens, PWL performs 20 random walk steps to change the theory, and then attempts `init_proof` again. If this fails, PWL repeats the process: perform another 20 iterations of random walk and then try `init_proof` again, etc. We find in our experiments that during proof initialization, `init_proof` returns *null* 72.5% of the time on the question-answering task of the ProofWriter dataset when using classical logic (not counting the invocations of `init_proof` by the fourth MH proposal in table 1). However, when we switch to intuitionistic logic, `init_proof` did not return *null* on the same dataset. This is due to the fact that under classical logic, the formula $A \; ! \; B$ is equivalent to $: A \_ B$, and so *init_proof* will attempt to prove either $: A$ or B is true, but this equivalence is not necessarily true under intu-

itionistic logic. The examples in ProofWriter contain many instances of $A \rightarrow B$ but not $A \vee B$.

MH can become stuck in regions of locally high probability, and unless it is run for significantly more iterations, it will be unable to find globally optimal regions of the space of theories and proofs. To help alleviate this, at every 100^{th} iteration, we perform 20 steps of a random walk: Each step is an MH step, only using the third and fourth proposal in the table 1, and every MH proposal is accepted regardless of its acceptance probability. This re-initialization is in many ways analogous to a random restart and can help to escape from local maxima.

We emphasize that while the generative process describes *deductive reasoning*, the inference algorithm is that of *abductive reasoning* (coupled with deductive reasoning for consistency checking).

3.3.1 *Computing the semantic prior* $p(x \mid x)$

An important quantity that PWL needs to compute is the *semantic prior* $p(x \mid x)$, which is the probability of a new logical form x given a set of previously observed logical forms x, $\{x_1, \ldots, x_n\}$. The quantity is needed when reading a sentence, in order to rerank the list of candidate logical forms, and is the principal way in which semantic information from the theory is incorporated during reading. PWL also computes this quantity when answering true/false or multiple-choice questions, since it can compare the probability of one logical form versus another. This expression can be written

$$p(x \mid x) = \frac{p(x_1, \ldots, x_n, x)}{p(x_1, \ldots, x_n)}. \tag{25}$$

The numerator and denominator are approximated with a sum over the possible theories T and proofs π:

$$p(x_1, \ldots, x_n) = \sum_{T, \pi} p(T) \prod_{i=1} \mathbf{1}\{\pi_i \text{ is a proof of } x_i\} p(\pi_i \mid T), \tag{26}$$

$$\sum_{\substack{T^{(t)}, \pi^{(t)} \text{ distinct} \\ \text{samples from } T, \pi \mid x}} p(T^{(t)}) \prod_{i=1} p(\pi_i^{(t)} \mid T^{(t)}), \tag{27}$$

$$p(x_1, \ldots, x_n, x) = \sum_{T, \pi, \pi} p(T) \mathbf{1}\{\pi \text{ is a proof of } x\} p(\pi \mid T) \prod_{i=1} \mathbf{1}\{\pi_i \text{ is a proof of } x_i\} p(\pi_i \mid T), \tag{28}$$

$$\sum_{\substack{T^{(t)}, \pi^{(t)}, \pi^{(t)} \text{ distinct} \\ \text{samples from } T, \pi, \pi \mid x}} p(T^{(t)}) p(\pi^{(t)} \mid T^{(t)}) \prod_{i=1} p(\pi_i^{(t)} \mid T^{(t)}). \tag{29}$$

Since the quantity in equations 26 and 28 are intractable to compute, PWL approximates them by sampling from the posterior $T, \ldots, \ldots \mid x_1, \ldots, x_n$ and summing over the distinct samples. Although this approximation seems crude, the sum is dominated by a small number of the most probable theories and proofs, and MH is an effective way to find them, as we observe in experiments. But it may be promising to explore other approaches to compute this quantity, such as Luo et al. (2020). In many applications, we do not need to compute the denominator. For example, if we wish to determine which of two logical forms is more probable, x or x_+, given the previously observed logical forms x, we can approximate the ratio

$$\frac{p(x \mid x_1, \ldots, x_n)}{p(x_+ \mid x_1, \ldots, x_n)} = \frac{p(x_1, \ldots, x_n, x)}{p(x_1, \ldots, x_n, x_+)}, \tag{30}$$

using the sampling procedure. The term $p(x_1, \ldots, x_n)$ appears in both the numerator and denominator, and so it cancels.

3.4 KEY DESIGN CHOICES AND FUTURE DIRECTIONS

Many of the design choices in the design and implementation of the reasoning module were made for the ease of implementation and rapid prototyping. As a result, there is significant room for improvement on many aspects of this module. For example, the prior on the theory $p(T)$ is very simple. The real world has much richer structure, with an ontology of types. A better prior for T would explicitly generate this hierarchical ontology, along with mutual exclusion and subsumption relationships.

While our prior for entity names prefers that each entity has a small number of names (usually 1), it does not prefer that each name refers to a small number of entities. A better choice of prior would be one where the mapping between entities and names is closer to one-to-one. An even better approach would be to alter the prior on the theory $p(T)$ so that with some probability, each generated relation is one-to-one (or very close to one-to-one). The `name` relation would be specified as one-to-one, and we would not need a separate prior for entity names.

The largest bottleneck in PWL is consistency checking, performed by `provable` (algorithm 1) along with its helper functions. These algorithms take into account every axiom in the theory, regardless of whether or not the axiom is related to the formula argument A. Finding a way to relax this would substantially improve the scalability to larger theories that contain many more axioms. For instance, `provable` could be modified to only consider axioms that are sufficiently relevant to the formula argument A. This would require appropriately defining "relevance" and to be permissive of at least *some* inconsistencies. One natural question that arises is whether the inconsistencies should be part of the model or a consequence of approximate inference. Furthermore, `provable` currently does not consider all possible

proofs of the given formula A. While this sufficed in our experiments, there may be cases where it does not. Further exploration is needed to identify such cases and to find ways to extend `provable` to consider additional proof paths as appropriate.

Related to this issue is the algorithm for checking the consistency of set sizes (algorithm 5). This algorithm has exponential running time in the worse-case, even though it always terminates quickly in our experiments. Further exploration is needed here, as well, to find cases where the running-time is problematic. In the same vein as consistency checking above, one possible way to resolve the worst-case complexity issue is to modify the algorithm to only consider a small number of "relevant" sets, rather than all known sets in the theory. In addition, PWL does not find all possible inconsistencies in set sizes. For example, consider the collection of set size axioms `size( X.cat(X)) = 3`, `size( X.small(X)) = 2`, and `size( X(cat(X) _ small(X))) = 10`. Again due to the fact that jA [Bj = jAj + jBj - jA \ Bj for any two sets A and B, it must be the case that the set `X(cat(X) _ small(X))` has size at most 5. But algorithm 5 will only provide an upper bound on the size of a set A if that set is a subset of another set A S (and is therefore part of a clique of subsets of S that are mutually disjoint). Future work to extend the consistency checking of set sizes to handle the above case is welcome.

The prior for the proofs p(¡ j T) is very simple. The premises of each proof step are sampled uniformly at random from the set of available logical forms, which grows linearly with the size of the proof. Thus, longer proofs will be unfairly penalized by this prior. This was not a problem in our experiments as the proofs tended to be fairly short. A more realistic prior would be more directed and context-aware. For example, if a logical form is being generated for a sentence that is part of a conversation about astronomy, then the next logical form is more likely to utilize constants and axioms from the domain of astronomy. A more realistic prior would also generate reusable proof fragments, which can be used multiple times across proofs. An alternate approach for generating proofs could be to use a compositional exchangeable distribution such as adaptor grammars (Johnson, Griffiths, and Goldwater, 2006).

The `init_proof` function (algorithm 6) attempts to find a proof of a given logical form (creating axioms as needed) without any modifications to existing axioms in the theory. As a result, it may fail to a find a proof if there is an axiom that is inconsistent with the given logical form, even if the logical form is consistent with all other observed logical forms. While we described a workaround above, the random walk approach is unfocused: and for large theories, it is unlikely to change the specific axioms that are inconsistent with the given logical form. A better approach would be to focus the random walk on these axioms.

Perhaps a better approach is to allow `init_proof` to change existing axioms to accommodate the new proof.

The first MH proposal in table 1 is simple but restrictive: the antecedent conjuncts and the consequent are restricted to be atomic. The inference would be able to explore a much larger and semantically richer set of theories if the antecedent or consequent could contain more complex formulas, including other quantified formulas. In addition, the inference algorithm sometimes becomes stuck in local maxima, requiring more MH iterations to find more global maxima. One way to improve the efficiency of inference is to add a new MH proposal that specifically proposes to split or merge types. For example, if the theory has the axioms $\texttt{cat}(\texttt{c}_1)$ and $\texttt{dog}(\texttt{c}_1)$, this proposal would split $\texttt{c}_1$ into two concepts so that $\texttt{cat}(\texttt{c}_1)$ and $\texttt{dog}(\texttt{c}_2)$ are axioms. Without this new proposal, this transformation is still reachable via successive applications of the 4$^{\text{th}}$ proposal in table 1, but if both axioms are used in multiple proofs each, the intermediate proposals would have very low probability. This kind of type-based Markov chain Monte Carlo is similar in principle to Liang, Jordan, and Klein (2010).

The MH proposal distribution in PWL is very naive, picking a proposal almost uniformly at random. This is potentially very wasteful, especially when the number of proofs is high and/or the proofs are large. A better algorithm would be aware of the task at hand. For example, if the current task is to answer a question about geography, the MH proposals should focus on proofs of logical forms related to geography, and very rarely select a proof of a logical form in an unrelated domain.

The experiments did not have any situations where there was significant uncertainty in the theory, and so it sufficed to use a single Markov chain for inference. However, if the true posterior of the theory is multi-modal, a single Markov chain might only be able to find one of the modes, especially if the regions between the modes have low probability. Using multiple Markov chains would provide a more robust approximation of the posterior.

The following list summarizes the key design choices discussed in this chapter:

- The prior for the theory $\texttt{p}(\texttt{T})$ was chosen to be fairly simple and flat in structure. Even so, it has the property that larger and more complex theories have lower probability (Occam's razor). While this worked sufficiently well in our experiments, a richer prior, such as one that includes an explicit ontology, would be preferable.

- However, the theory prior is not so simple as to treat entity names and sets in the same way as it treats other objects. The prior on entity names was chosen in order to encourage each entity to have a small number of names, and this aligns with our assumption that the name relation is almost one-to-one.

- The reasoning module contains a specialized "submodule" for reasoning about sets, and a data structure to facilitate this. We chose to do so since a large part of reasoning in natural language is reasoning over collections of objects. Language has many built-in features that enable the seamless communication of information about such collections.

- The set reasoning component, along with the `provable` function (algorithm 1) and its helper functions, provides PWL with a way to check for the consistency of the theory. This does not search over all possible proofs, as such an approach would never terminate (e.g. `provable_by_exclusion` in algorithm 2 explicitly avoids searching for nested proofs by exclusion). But the coverage of this approach is evidently sufficient to find the contradictions that arise when reading natural language sentences in our experiments.

- The consistency checking is unfocused: when checking for the consistency of a new axiom, PWL currently attempts to find inconsistencies with respect to every other axiom in the theory, no matter how unrelated. This approach is computationally expensive but conceptually simple, since we can avoid questions about how to measure "relatedness" and whether logical inconsistencies are part of the model or a consequence of approximate inference.

- The prior for the proofs $p(\ \mid T)$ was also chosen to be fairly simple. The premises for each proof step are sampled uniformly at random from the conclusions of the previous proof steps, which while simple, is highly unrealistic. Human reasoning is much more directed, and relies on common proof fragments that are re-used across proofs.

- The proof initialization function `init_proof` (algorithm 6) attempts to find a proof for the given logical form, creating new axioms as needed. Its recursive structure over the expression tree of the logical form allows `init_proof` to handle a wide variety of logical forms, while keeping the axioms simple. A naive alternative would be to simply add the given logical form as an axiom, but this would only move the complexity from proof initialization to the MH proposals. `init_proof` does not consider proofs that utilize existing theorems in the theory.

- In addition, `init_proof` does not try to modify already existing axioms in order to make the theory consistent with the new logical form. In this case, we perform 20 steps of a random walk to modify the existing axioms in the theory, with the hope that they become consistent with the new logical form. This

procedure is unguided, and a better alternative would be to focus on changing the specific axioms that led to the inconsistency.

- In `init_proof`, the `swap` function (on line 26) reorders the possible instantiations of an existential quantifier in order to produce a proof with higher prior probability, thereby reducing the number of MH iterations needed to find a good theory and proofs.

- PWL uses a fairly simple set of MH proposals (listed in table 1). The fourth proposal in the list was designed to explore the other possible proofs that may be constructed by the `init_proof` function. It is also fairly easy to implement since it can use a slightly modified version of the `init_proof` function. However, this proposal will select proof nodes uniformly at random to resample, regardless of the size of the proof at that node. `init_proof` will then proceed to try resampling the full proof. If the selected proof is large, it may require many attempts of `init_proof` to find a new proof. An alternate approach that only resamples fragments of proofs may perform better.

- PWL has a MH proposal specifically to change the sizes of sets. This proposal is not strictly necessary since the fourth proposal can achieve the same thing, but to do so would require more iterations, since it would need to select the appropriate proof node. This proposal also helps to ensure that MH is able to sufficiently explore the space of possible set sizes. However, if the selected set is equivalent to another known set, then our algorithms for finding the upper and lower bound for the size of the selected set will return the same size, essentially wasting an MH step. To avoid this, this step should instead simultaneously change the sizes of all sets that are equivalent to the selected set. In our graph-based data structure for maintaining the consistency of set sizes, this entails finding the strongly-connected component that contains the vertex that corresponds to the selected set, contracting the component into a single vertex, and then continuing with our algorithms to find the lower and upper bound on the size of the set that corresponds to this vertex.

- PWL also includes a split and merge proposals for MH, where it proposes to merge repeated events in the theory, or split an event into two. This is helpful in a case such as when two entities have the same name. The merge operation will propose merging these two entities into a single entity (and the two name events into a single event). In principle, repeated applications of the fourth MH proposal could produce the same result, but it would require more iterations. The split proposal is necessary to ensure that the MH acceptance probability is not 0, but the probability

of proposing a split is set very low, since the prior on the theory $p(T)$ favors smaller theories.

- At every 100^{th} iteration of MH, we perform 20 steps of a random walk, akin to a "random restart" in optimization. This helps MH to escape regions of locally high probability and hopefully find regions of globally high probability.
- We chose to compute the semantic prior by using MH to find distinct high-probability samples of the theory and proofs. MH is an effective way to find high-probability regions of the space of theories and proofs.

4

LANGUAGE MODULE

In the previous chapter, we described the reasoning module, which constitutes one half of PWL. In this chapter, we present the other half: the language module. This module governs the relationship between the logical forms and the natural language utterances. A mathematical description of the model is provided in section 4.2. In section 4.3.1, we describe the algorithms for training and parsing, including details on their implementation. In section 4.4, we apply this parsing approach to the GeoQuery and Jobs datasets (Tang and Mooney, 2000; Zelle and Mooney, 1996), using the Datalog representation of the provided logical form labels, and demonstrate that the accuracy of the parsed logical forms is comparable to that of the state-of-the-art on these datasets. Since the Datalog representation in these datasets are fairly domain-specific, we present a new wide-coverage semantic representation based on higher-order logic in section 4.5.

Accurate and efficient semantic parsing is a long-standing goal in natural language processing. Existing approaches are quite successful in particular domains (Dong and Lapata, 2016; Kwiatkowski et al., 2013, 2010, 2011; Li, Liu, and Sun, 2013; Liang, Jordan, and Klein, 2013; Rabinovich, Stern, and Klein, 2017; Wang, Kwiatkowski, and Zettlemoyer, 2014; Wong and Mooney, 2007; Zettlemoyer and Collins, 2005, 2007; Zhao and Huang, 2015). However, they are largely domain-specific, relying on additional supervision such as a lexicon that provides the semantics or the type of each token in a set (Dong and Lapata, 2016; Kwiatkowski et al., 2010, 2011; Liang, Jordan, and Klein, 2013; Rabinovich, Stern, and Klein, 2017; Wang, Kwiatkowski, and Zettlemoyer, 2014; Zettlemoyer and Collins, 2005, 2007; Zhao and Huang, 2015), or a set of initial synchronous context-free grammar rules (Li, Liu, and Sun, 2013; Wong and Mooney, 2007). To apply the above systems to a new domain, additional supervision is necessary. When beginning to read text from a new domain, humans do not need to re-learn basic English grammar. Rather, they may encounter novel terminology. With this in mind, our approach is akin to that of (Kwiatkowski et al., 2013) where we provide domain-independent supervision to help train a semantic parser. More specifically, PWL restricts the rules that may be learned during training to a set that characterizes the general syntax of English. While we do not explicitly present and evaluate

an open-domain semantic parser, we hope our work provides *a step* in that direction.

Knowledge plays a critical role in natural language understanding. Even seemingly trivial sentences may have a large number of ambiguous interpretations. Consider the sentence *"Ada started the machine with the GPU,"* for example. Without additional knowledge, such as the fact that "machine" can refer to computing devices that contain GPUs, or that computers generally contain devices such as GPUs, the reader cannot determine whether the GPU is part of the machine or if the GPU is a device that is used to start machines. Context is highly instrumental to quickly and unambiguously understand sentences.

In contrast to most semantic parsers, which are built on discriminative models, our model is fully generative: To generate a sentence, the logical form is first drawn from a prior. A grammar then recursively constructs a derivation tree top-down, probabilistically selecting production rules from distributions that depend on the logical form. The generative nature of the semantic parsing model allows it to fit seamlessly into our larger model. The semantic prior distribution provides a straightforward way to incorporate background knowledge, such as information about the types of entities and predicates, or the context of the utterance. In fact, to fit this semantic parsing model into our larger model, the semantic prior is simply replaced with our distribution of logical forms conditioned on the theory $p(\ _i \mid T)$. Additionally, our generative model presents a promising direction to *jointly* learn to understand and generate natural language. In addition, our parser can return *partial* parses of sentences, which is useful for sentences that contain a small number of unseen words, such as definitions of new tokens. This can be exploited to learn new tokens and concepts outside of training.

4.1 BACKGROUND: HIERARCHICAL DIRICHLET PROCESSES

Before introducing the model for the language module in the next section, we first provide background on performing inference in Dirichlet processes using Gibbs sampling, and on hierarchical Dirichlet processes, which constitute a central component in the language module of PWM. Later in this section, we present a novel application of the HDP to model distributions that depend on discrete structures, such as sequences, tree, logical forms, etc. See section 3.1 for background on the definition of Dirichlet processes and Chinese restaurant processes.

GIBBS SAMPLING FOR DIRICHLET PROCESSES: The *Chinese restaurant process* (CRP) representation of the Dirichlet process enables efficient inference using *Markov chain Monte Carlo* (MCMC) methods. Suppose that we are given $y\ , \{y_1, \ldots, y_n\}$ observations and we wish to infer the values of the latent variables: $\ _i$ and z_i (using the same

notation as section 3.1). A Gibbs sampling algorithm can be derived, where initial values for θ_i and z_i are selected, $\theta_i^{(0)}$ and $z_i^{(0)}$, and for each iteration t, we sample new values of $\theta_i^{(t)}$ and $z_i^{(t)}$. One straightforward initialization for $\theta_i^{(0)}$ and $z_i^{(0)}$ is to assign each observation to its own table: $\theta_i^{(0)} = y_i$ and $z_i^{(0)} = i$ for $i = 1, \ldots, n$. Note that the value of $\theta_i^{(t)}$ is deterministic and equal to $y_{z_j^{(t)}}$ for all $z_j^{(t)} = i$. Thus, only $z_i^{(t)}$ needs to be sampled at each iteration (for all $i = 1, \ldots, n$). In Gibbs sampling, each random variable is sampled from its conditional distribution given all other variables: $z_i^{(t+1)} \sim z_i \mid \theta^{(t)}, z_{-i}^{(t)}, y$.

$$p(z_i \mid \theta, z_{-i}, y) \propto p(z, \theta, y), \tag{31}$$

$$= p(z_1, \ldots, z_n) \prod_{j=1}^{\infty} p(\theta_j) \prod_{j=1}^{n} p(y_j \mid \theta, z_j), \tag{32}$$

$$\propto p(z_{\theta(1)}, \ldots, z_{\theta(n)}) \mathbf{1}\{y_i = \theta_{z_i}\}, \tag{33}$$

$$p(z_i = k \mid \theta, z_{-i}, y) \propto \begin{cases} \mathbf{1}\{y_i = \theta_k\} \frac{n_k}{\alpha + n - 1} & \text{if } n_k > 0, \\ \mathbf{1}\{y_i = \theta_k\} \frac{\alpha\, p(\theta_k = y_i)}{\alpha + n - 1} & \text{if } n_k = 0, \end{cases} \tag{34}$$

where n_k is the number of customers sitting at table k not including the i^{th} customer, $\theta = \{\theta_1, \theta_2, \ldots\}$ and $z_{-i} = z \setminus \{z_i\}$ is the set of all z_j *except* z_i, and $\mathbf{1}\{\cdot\}$ is 1 if the condition is true and zero otherwise. In this derivation, we used exchangeability to change the order of the table assignments z so that z_i is the last assignment. After sufficiently many iterations, the distribution of the samples $\theta_i^{(t)}$ and $z_i^{(t)}$ will approach the true posterior $p(\theta, z \mid y)$.

Note that this presentation of the DP differs from the classical presentation, where the DP is part of a mixture model, as in:

$$G \sim \mathrm{DP}(\alpha, H), \tag{35}$$

$$\theta_1, \theta_2, \ldots \sim G, \tag{36}$$

$$y_i \sim F(\theta_i), \tag{37}$$

where $F(\theta_i)$ is a distribution with parameter θ_i. If H is a conjugate prior of F, then an efficient Gibbs sampling algorithm is available, for example if H is a Dirichlet distribution and F is a multinomial, or if both H and F are normal distributions. In this thesis, F is assumed to be the delta function (the distribution whose samples are identical to the input parameter), and no assumptions are made on H other than there exists an efficient way to compute the prior probability $p(\theta_i)$.

4.1.1 *Hierarchical Dirichlet processes*

The DP can be used as a component in larger models. The *hierarchical Dirichlet process* (HDP) (Teh et al., 2006) is a hierarchy of random variables, where each random variable is a distributed according to a Dirichlet process whose base distribution is given by the parent node

in the hierarchy. Suppose each observation y_i is coupled with a parameter x_i that indicates the source node from which to sample the observation. Let the label of the root node in the hierarchy be $\mathbf{o}$, and the model can be written:

$$G^{\mathbf{n}} \begin{cases} \sim \mathrm{DP}(\ ^{\mathbf{o}}, H) & \text{if } \mathbf{n} = \mathbf{o}, \\ \sim \mathrm{DP}(\ ^{\mathbf{n}}, G^{\mathrm{parent}(\mathbf{n})}) & \text{otherwise,} \end{cases} \tag{38}$$

$$y_i \sim G^{x_i}, \tag{39}$$

for all nodes in the hierarchy $\mathbf{n}$. An equivalent "Chinese restaurant" representation may be written, which is coined a *Chinese restaurant franchise* (CRF), where each node $\mathbf{n}$ has a restaurant. For simplicity, assume that all x_i are leaf nodes, then the CRF is written:

$$\ ^{1}, \ ^{2}, \ldots \sim H, \tag{40}$$

$$z_1^{\mathbf{n}} = 1, \tag{41}$$

$$z_{i+1}^{\mathbf{n}} = \begin{cases} k & \text{with probability } \frac{n_k^{\mathbf{n}}}{\ ^{\mathbf{n}} + i}, \\ k^{\mathrm{new}} & \text{with probability } \frac{\ ^{\mathbf{n}}}{\ ^{\mathbf{n}} + i}, \end{cases} \tag{42}$$

$$\ ^{\mathbf{n}}_i = \begin{cases} z_i^{\mathbf{o}} & \text{if } \mathbf{n} = \mathbf{o}, \\ z_i^{\mathrm{parent}(\mathbf{n})} & \text{otherwise,} \end{cases} \tag{43}$$

$$y_i = \ ^{x_i}_{u_i + 1}, \tag{44}$$

for all nodes in the hierarchy $\mathbf{n}$, where $n_k^{\mathbf{n}} \triangleq \#\{j < i : z_j^{\mathbf{n}} = k\}$ is the number of customers at node $\mathbf{n}$ sitting at table k, $k^{\mathrm{new}} \triangleq \max\{z_1^{\mathbf{n}}, \ldots, z_i^{\mathbf{n}}\} + 1$ is the next available table at node $\mathbf{n}$, and $u_i \triangleq \#\{j < i : x_j = x_i\}$ is the number of previous observations drawn from node $\mathbf{n}$. In this extended metaphor, whenever a customer sits at a new table in the restaurant at node $\mathbf{n} \neq \mathbf{o}$, a "new customer" appears in the parent node $\mathrm{parent}(\mathbf{n})$ which corresponds to this table. The $\ ^{\mathbf{n}}_i$ are the samples from $G^{\mathbf{n}}$. Note that the above model is valid only when x_i is a leaf node. If x_i were a parent node, then the output samples $\ ^{x_i}_j$ are used by both the child nodes of x_i as well as the observations y_i. In the restaurant metaphor, the customers at node x_i not only come from its child nodes but also from the observations. In this case, the $\ ^{x_i}_j$ that are assigned to the observations come after those assigned to child nodes (the order does not actually matter thanks to exchangeability, so long as the samples/customers are partitioned between the two). More precisely, y_i would be equal to $\ ^{x_i}_{c^{\mathbf{n}} + u_i + 1}$ where $c^{\mathbf{n}} = \max\{c_i^{\mathbf{c}} : \mathbf{c} \in \mathrm{children}(\mathbf{n})\}$ is the number of $\ ^{x_i}_j$ used by the child nodes of $\mathbf{n}$ (i.e. the number of customers that come from the child nodes of $\mathbf{n}$).

INFERENCE: The Gibbs sampling update can be derived similarly to the DP case: Given (x, y, z), and can be computed deterministically. Thus we only need to sample each $z_i^{\mathbf{n}}$:

$$p(z_i^{\mathbf{n}} \mid , , z^{-\mathbf{n}}, z^{\mathbf{n}}_{-i}, x, y) \propto p(z, , , x, y), \tag{45}$$

$$= \prod_{j=1} p(_j) \prod_{\mathbf{n}} p(z_1^{\mathbf{n}}, z_2^{\mathbf{n}}, \ldots) \prod_{j=1} p(_j^{\mathbf{n}} \mid , ^{-\mathbf{n}}, z_j^{\mathbf{n}}) \prod_{j=1} p(y_j \mid , x_j), \tag{46}$$

$$\propto \begin{cases} p(z^{\mathbf{o}}_{(1)}, z^{\mathbf{o}}_{(2)}, \ldots) 1\{ _i^{\mathbf{o}} = z_i^{\mathbf{o}}\} & \text{if } \mathbf{n} = \mathbf{o}, \\ p(z^{\mathbf{n}}_{(1)}, z^{\mathbf{n}}_{(2)}, \ldots) 1\{ _i^{\mathbf{n}} = z_i^{\mathrm{parent}(\mathbf{n})}\} & \text{otherwise,} \end{cases} \tag{47}$$

$$p(z_i^{\mathbf{n}} = k \mid , , z^{-\mathbf{n}}, z^{\mathbf{n}}_{-i}, x, y) \propto \tag{48}$$

$$\begin{cases} 1\{ _i^{\mathbf{o}} = k\} \frac{n_k^{\mathbf{o}}}{ ^{\mathbf{o}} + n^{\mathbf{o}}} & \text{if } \mathbf{n} = \mathbf{o}, n_k^{\mathbf{n}} > 0, \\ 1\{ _i^{\mathbf{o}} = \mathrm{new}\} \frac{p(_{\mathrm{new}} = _i^{\mathbf{o}}) \ ^{\mathbf{o}}}{ ^{\mathbf{o}} + n^{\mathbf{o}}} & \text{if } \mathbf{n} = \mathbf{o}, n_k^{\mathbf{n}} = 0, \\ 1\{ _i^{\mathbf{n}} = _k^{\mathrm{parent}(\mathbf{n})}\} \frac{n_k^{\mathbf{n}}}{ ^{\mathbf{n}} + n^{\mathbf{n}}} & \text{if } \mathbf{n} \neq \mathbf{o}, n_k^{\mathbf{n}} > 0, \\ 1\{ _i^{\mathbf{n}} = _{\mathrm{new}}^{\mathrm{parent}(\mathbf{n})}\} \frac{f^{\mathrm{parent}(\mathbf{n})}(_i^{\mathbf{n}}) \ ^{\mathbf{n}}}{ ^{\mathbf{n}} + n^{\mathbf{n}}} & \text{if } \mathbf{n} \neq \mathbf{o}, n_k^{\mathbf{n}} = 0, \end{cases} \tag{49}$$

where $n_k^{\mathbf{n}}$ is the number of customers at node $\mathbf{n}$ sitting at table k *not including* the customer currently being resampled, $n^{\mathbf{n}}$ is the total number of customers at node $\mathbf{n}$ (also not including the current customer), and $f^{\mathbf{m}}(v)$ is shorthand for $p(_{\mathrm{new}}^{\mathbf{m}} = v \mid , , z^{-\mathbf{n}}, z^{\mathbf{n}}_{-i}, x, y)$ for any node $\mathbf{m}$. Note that in the case where $\mathbf{n} \neq \mathbf{o}$, sampling $z_i^{\mathbf{n}}$ requires computing $f^{\mathrm{parent}(\mathbf{n})}(_i^{\mathbf{n}})$, i.e. the probability of a customer at node $\mathbf{n}$ choosing to sit a "new" table $p(_{\mathrm{new}}^{\mathrm{parent}(\mathbf{n})})$. This value can be computed recursively, so if the node $\mathbf{m} \neq \mathbf{o}$:

$$f^{\mathbf{m}}(v) = \frac{ ^{\mathbf{m}} f^{\mathrm{parent}(\mathbf{m})}(v)}{ ^{\mathbf{m}} + n^{\mathbf{m}}} + \sum_{k' : n_{k'}^{\mathbf{m}} > 0} \frac{n_{k'}^{\mathbf{m}} 1\{ _{k'}^{\mathrm{parent}(\mathbf{m})} = v\}}{ ^{\mathbf{m}} + n^{\mathbf{m}}}. \tag{50}$$

In the case where $\mathbf{m} = \mathbf{o}$:

$$f^{\mathbf{o}}(v) = \frac{ ^{\mathbf{o}} p(_{\mathrm{new}} = v)}{ ^{\mathbf{o}} + n^{\mathbf{o}}} + \sum_{k' : n_{k'}^{\mathbf{o}} > 0} \frac{n_{k'}^{\mathbf{o}} 1\{ _{k'} = v\}}{ ^{\mathbf{o}} + n^{\mathbf{o}}}. \tag{51}$$

If $z_i^{\mathbf{n}}$ is sampled to be a "new" table, a new customer will appear in the parent node of $\mathbf{n}$, and its table assignment must be sampled next. This new customer may itself be assigned to a new table, and so this process continues recursively until a customer sits at a non-empty table, or a customer sits at an empty table at the root node $\mathbf{o}$. The computation required in this recursive sampling procedure overlaps heavily with that in computing the probabilities in equations 50 and 51, so they should be done simultaneously to avoid wasted computation.

In our code, for each iteration of Gibbs sampling, we traverse the tree nodes $\mathbf{n}$ in prefix order, and resample $z_i^{\mathbf{n}}$ in random order.

In many applications, including in PWL, we need to compute the probability of a new observation y_{n+1}, given its source node x_{n+1} and previous observations (x, y):

$$p(y_{n+1} \mid x_{n+1}, x, y) = \int p(y_{n+1} \mid x_{n+1}, z) p(z \mid x, y) dz, \tag{52}$$

$$\frac{1}{N_{\text{samples}}} \sum_{z^{(t)} \sim p(z \mid x, y)} p(y_{n+1} \mid x_{n+1}, z^{(t)}, \ldots, \ldots). \tag{53}$$

The integral is approximated as a sum over posterior samples of z, which can be obtained using the MCMC algorithm described above. However, we find in our experiments that the posterior is concentrated at a single point, and it suffices to keep only the final sample (i.e. $N_{\text{samples}} = 1$) as a point estimate of $z, \ldots, \ldots$. In either case, we can compute the quantity within the sum:

$$p(y_{n+1} \mid x_{n+1}, z, \ldots, \ldots) = p(x^{n+1}_{\text{new}} = y_{n+1} \mid z, \ldots, \ldots). \tag{54}$$

This quantity can be computed as in equations 50 and 51 (but since we are not resampling z^n_i, we don't exclude any customers in the n^m_k terms).

The above can be extended to the case where rather than 1 new observation, there are k new observations, and we want to compute their joint probability:

$$p\left(\bigcap_{i=1}^{k} y_{n+i} \,\middle|\, x, y, \bigcap_{i=1}^{k} x_{n+i}\right) = \prod_{i=1}^{k} p\left(y_{n+i} \,\middle|\, x, y, \bigcap_{j=1}^{i} x_{n+j}, \bigcap_{j=1}^{i-1} x_{n+j}\right). \tag{55}$$

So to compute this, first compute the probability of the first observation y_{n+1} alone. Next, add (x_{n+1}, y_{n+1}) to the HDP (treat them as part of x and y) and compute the probability of y_{n+2} alone. Repeat until all k probabilities are computed and then return the product. We observe that the joint probability does not factorize over each observation. This is due to the "rich get richer" effect observed in the Chinese restaurant process: If one observation is sampled, the same observation is more likely to be sampled in the future, since future customers are more likely to sit at tables with existing customers. And so the distribution is not i.i.d.

But as the number of customers n becomes very large, the effect of and any single customer on the distribution of the next observation becomes negligible, and so the distribution becomes more i.i.d.:

$$\lim_{n \to \infty} p\left(\bigcap_{i=1}^{k} y_{n+i} \,\middle|\, x, y, \bigcap_{i=1}^{k} x_{n+i}\right) = \lim_{n \to \infty} \prod_{i=1}^{k} p\left(y_{n+i} \,\middle|\, x, y, \bigcap_{i=1}^{k} x_{n+i}\right). \tag{56}$$

This fact can be useful when approximating

$$p\left(\bigcap_{i=1}^{k} y_{n+i} \,\middle|\, x, y, \bigcap_{i=1}^{k} x_{n+i}\right) \quad \prod_{i=1}^{k} p\left(y_{n+i} \,\middle|\, x, y, \bigcap_{i=1}^{k} x_{n+i}\right), \quad (57)$$

when n is large.

LEARNING THE CONCENTRATION PARAMETER : We learn the concentration parameter from the data by placing a Gamma prior on :

$$^{n} \quad \text{Gamma}(a^{n}, b^{n}). \quad (58)$$

An auxiliary variable sampling method can be used to infer , which is described in appendix A of Teh et al. (2006) and section 6 of Escobar and West (1995). The Gibbs sampling step for ^{n} is:

$$s^{n} \quad \text{Bernoulli}\left(\frac{n^{n}}{^{n} + n^{n}}\right), \quad (59)$$

$$w^{n} \quad \text{Beta}(\ ^{n} + 1, n^{n}), \quad (60)$$

$$^{n} \quad \text{Gamma}(a^{n} + \text{maxfz}_{i}^{n}\text{g} - s^{n}, b^{n} - \log w^{n}). \quad (61)$$

Here, $\text{maxfz}_{i}^{n}\text{g}$ is the number of occupied tables in restaurant $\mathbf{n}$. The above updates assume that each node $\mathbf{n}$ has an independent ^{n}. However, in many scenarios, we wish to tie the concentration parameters together to improve statistical efficiency. Suppose we constrain all the concentration parameters at each level in the hierarchy to be equal. Let $L(\mathbf{n})$ be defined as the level of the node $\mathbf{n}$ (i.e. $L(o) = 0$ and $L(\mathbf{n}) = L(\text{parent}(\mathbf{n})) + 1$). Let $_{i}$ be the concentration parameter at level i, and so $^{n} = {}_{L(\mathbf{n})}$. Let its prior be $_{i}$ Gamma(a_i, b_i). Then, the Gibbs sampling step for $_{i}$ is:

$$_{i} \quad \text{Gamma}\left(a_i + \sum_{\text{fn}:L(\mathbf{n})=i\text{g}} (\text{maxfz}_{i}^{n}\text{g} - s^{n}), b_i - \sum_{\text{fn}:L(\mathbf{n})=i\text{g}} \log w^{n}\right). \quad (62)$$

This is the approach we implement in PWL when training the language module. PWL has several HDP hierarchies, and each has its own set of hyperparameters a and b. As an example, for one such hierarchy in PWL (corresponding to the nonterminal VP_R), the hyperparameters are $a_1 = 100, a_2 = 10, b_1 = 0.1, b_2 = 1$, but the other hierarchies have similar values for their hyperparameters.

4.1.2 *Inferring the source node* x

The above describes how to obtain posterior samples of z (and therefore, and), given a set of observations y_i and the corresponding nodes x_i from which they were sampled. But now consider the case

where the x are random variables, and we encounter a new observation y, but the source node x (from which y was sampled) is unknown, and we would like to infer it. That is, we would like to compute:

$$\arg\max_x p(x \mid y, x, y) = \arg\max_x p(x) \int_z p(y \mid x, z) p(z \mid x, y) dz, \quad (63)$$

$$\arg\max_x \frac{p(x)}{N_{\text{samples}}} \sum_{z^{(t)} \sim z \mid x, y} p(y \mid x, z^{(t)}, \ldots^{(t)}, \ldots^{(t)}). \quad (64)$$

$$\text{where } p(y \mid x, z, \ldots, \ldots) = p(x_{\text{new}} = y \mid z, \ldots, \ldots). \quad (65)$$

Again, this quantity is computed as in equations 50 and 51. The $\arg\max$ over this objective is a discrete optimization problem, which, if solved naïvely, would require computing the objective function for every node n in the tree. This is intractable if the tree is very large. Therefore, we present a branch-and-bound algorithm to perform this optimization efficiently.

Algorithm 8: Pseudocode for a generic brand-and-bound algorithm for k-best discrete optimization.

```
function branch_and_bound(objective function f, heuristic h, domain X)
    C is an empty list
    Q is an empty priority queue
    Q.push(X, 1)
    while Q not empty do
        (S, v) = Q.pop()
        if S = {x} is a singleton
            C.add(x, f(x))
        else
            (S_1, ..., S_n) = branch(S)
            for i = 1, ..., n do
                Q.push(S_i, h(S_i))
        /* check termination condition */
        if there are k elements in C with priority at least v
            break
    return C /* the k elements of X that maximize f */
```

Branch-and-bound (Land and Doig, 1960) is a method for solving discrete optimization problems. Pseudocode is shown in algorithm 8. Given an objective function f, heuristic h, and search space X, the algorithm returns the k-best elements of X that maximize the objective f. The algorithm requires that the heuristic h be an *upper bound* for f. That is, for any set S,

$$h(S) > \max_{x \in S} f(x). \quad (66)$$

The algorithm begins by considering the full search space X. A procedure called `branch` then partitions X into n disjoint subsets X_i (this procedure is specific to the optimization problem). Each subset is

pushed onto the priority queue, with its key given by the heuristic $h(X_i)$. Then, for each iteration of the main loop, pop a set S from the priority queue, and repeat the process: using `branch`, partition S into $(S_1, \ldots, S_n)$, and then push each subset into the priority with key $h(S_i)$. If $S = \{x\}$ is a singleton set only containing the element x, then add it to a list of potential solutions. The algorithm terminates when there are k potential solutions whose objective function values are at least the priority of S, or when the priority queue becomes empty. Once the algorithm terminates, the objective function values of the returned solutions are at least as large as the heuristic of the remainder of the search space. And since h is an upper bound for f, the returned solutions are guaranteed to be optimal.

We develop a branch-and-bound algorithm to perform the optimization in equation 64. The HDP hierarchy provides a convenient search tree structure for the optimization. Let $D(\mathbf{n})$ be the set of descendent nodes of $\mathbf{n}$, including $\mathbf{n}$ itself. The function $\texttt{branch}(D(\mathbf{n}))$ is defined to partition $D(\mathbf{n})$ into $(\{\mathbf{n}\}, D(\mathbf{c}_1), \ldots, D(\mathbf{c}_n))$ where $\mathbf{c}_i$ are the child nodes of $\mathbf{n}$. We define a heuristic for $D(\mathbf{n})$:

$$h(D(\mathbf{n})) = \frac{h_x(D(\mathbf{n}))}{N_{\text{samples}}} \sum_{t=1}^{N_{\text{samples}}} \max_{k : n_k^{(t)} > 0} \{1\{z_{\mathbf{n}} = k\}, p(x_{\text{new}}^{\mathbf{n}} = y)\} \quad (67)$$

where $h_x(S)$ is an upper bound on the prior $h_x(S) > \max_{x \in S} p(x)$, the max is taken over all occupied tables in the restaurant at node $\mathbf{n}$, and the references to z within the sum are for the t^{th} sample, $z^{(t)}$. $D(\mathbf{n})$ can be sparsely represented in the implementation as a simple pointer to $\mathbf{n}$. The heuristic is convenient since it can be computed only using the information available at node $\mathbf{n}$, and so its running time is not a function of the size of the HDP hierarchy, as long as the heuristic on the prior $h_x(\,)$ is easy to compute. Furthermore, our algorithm avoids the recursion in the computation of $p(x_{\text{new}}^{\mathbf{n}})$, since the term $p(x_{\text{new}}^{\text{parent}(\mathbf{n})})$ was already computed in the computation of the heuristic for the *parent node*, and our algorithm re-uses it in future heuristic evaluations.

Thm 1. *The heuristic* $h(D(n))$ *is an upper bound on* $\max_{x \in D(n)} f(x)$ *where* f *is the objective function given by equation 64.*

Proof. Consider any node $\mathbf{m} \in D(\mathbf{n})$ a descendant of $\mathbf{n}$, and any MCMC sample t. We first aim to show that the quantity within the sum is an upper bound:

$$\max_{k : n_k^{(t)} > 0} \{1\{z_{\mathbf{n}} = k\}, p(x_{\text{new}}^{\mathbf{n}} = y)\} > p(y \mid x = \mathbf{m}, z^{(t)}, \theta^{(t)}, \phi^{(t)}). \quad (68)$$

Since the right-hand side is equal to $p(\ {}^{\mathbf{m}}_{\text{new}} = y\ )$, the bound is trivially true in the case where $\mathbf{m} = \mathbf{n}$. So we can assume without loss of generality that $\mathbf{m} \neq \mathbf{n}$, and the right-hand side can be written:

$$p(y \mid x = \mathbf{m}, z^{(t)}, \ {}^{(t)}, \ {}^{(t)}) = p(\ {}^{\mathbf{m}}_{\text{new}} = y\ ), \tag{69}$$

$$= \frac{{}^{\mathbf{m}}p(\ {}^{\text{parent}(\mathbf{m})}_{\text{new}} = y\ )}{{}^{\mathbf{m}} + n^{\mathbf{m}}} + \sum_{k' : n^{\mathbf{m}}_{k'} > 0} \frac{n^{\mathbf{m}}_{k'} 1\{\ {}^{\text{parent}(\mathbf{m})}_{k'} = y\ \}}{{}^{\mathbf{m}} + n^{\mathbf{m}}}, \tag{70}$$

according to equation 50. Since this expression is a convex combination of $1\{\ {}^{\text{parent}(\mathbf{m})}_{k'} = y\ \}$ and $p(\ {}^{\text{parent}(\mathbf{m})}_{\text{new}} = y\ )$, it is bounded above by:

$$\leq \max_{k' : n^{\mathbf{m}}_{k'} > 0} 1\{\ {}^{\text{parent}(\mathbf{m})}_{k'} = y\ \}, p(\ {}^{\text{parent}(\mathbf{m})}_{\text{new}} = y\ ) \ . \tag{71}$$

Due to equation 50, observe that $p(\ {}^{\mathbf{a}}_{\text{new}} = y\ ) \leq p(\ {}^{\text{parent}(\mathbf{a})}_{\text{new}} = y\ )$ for any node $\mathbf{a}$. In addition, by construction of the HDP, the ${}^{\mathbf{a}}_{k}$ at any node $\mathbf{a}$ are a subset of the ${}^{\text{parent}(\mathbf{a})}_{k}$. That is, for all k, there is a k' such that ${}^{\mathbf{a}}_{k} = {}^{\text{parent}(\mathbf{a})}_{k'}$. These observations extend to all ancestors of $\mathbf{a}$. Applying these two observations to the node $\mathbf{m}$, we can conclude that the above expression is further bounded above by:

$$\leq \max_{k' : n^{\mathbf{n}}_{k'} > 0} \{1\{\ {}^{\mathbf{n}}_{k'} = y\ \}, p(\ {}^{\mathbf{n}}_{\text{new}} = y\ )\}. \tag{72}$$

We have shown that the quantity within the sum of the heuristic is an upper bound. Since by definition, $h_x(D(\mathbf{n})) \geq \max_{x \in D(\mathbf{n})} p(x) \geq p(x = \mathbf{m})$, the full heuristic $h(D(\mathbf{n}))$ is an upper bound on $f(\mathbf{m})$, the objective function evaluated at $\mathbf{m}$, for all $\mathbf{m} \in D(\mathbf{n})$. Therefore, $h(D(\mathbf{n})) \geq \max_{\mathbf{m} \in D(\mathbf{n})} f(\mathbf{m})$.

The branch-and-bound algorithm starts with the input set $D(\mathbf{o})$, which is the set of all nodes in the tree, and will efficiently compute the k most probable values of the source node x, from which the observation y was sampled. Note that the above algorithm is easily extended to the case where the HDP is part of a mixture model (i.e. F is not a delta function). To do so, replace each instance of $1\{\ {}^{\mathbf{n}}_{k} = y\ \}$ with $p(y \mid y \sim F(\ {}^{\mathbf{n}}_{k}), z, \ )$, for all $\mathbf{n}$ and k.

The above algorithm can be generalized to the case where x is restricted to a subset of the nodes X in the hierarchy: $\arg\max_{x} p(x \mid x \in X, y, x, y)$. In this case, the algorithm is started with the input set $D(\mathbf{o}) \setminus X$. The `branch` function is modified: $\texttt{branch}(D(\mathbf{n}) \setminus X)$ partitions the set $D(\mathbf{n}) \setminus X$ into $(\{\mathbf{n}\} \setminus X, D(\mathbf{c}_1) \setminus X, \ldots, D(\mathbf{c}_n) \setminus X)$ where $\mathbf{c}_i$ are the child nodes of $\mathbf{n}$.

4.1.3 *Infinite hierarchies*

To apply the HDP in the language module of PWL, we need to be able to handle the case where the HDP hierarchy is infinite (but with

finite height). That is, every non-leaf node in the hierarchy may have an infinite number of children. But this makes no difference in the MCMC algorithm to infer z, , , since the number of given observations (x, y) is finite. We only need to compute and keep track of the variables that are associated with an observation (either at the current node or a descendant). Thus, the only nodes of the tree that we need to explicitly keep in memory are those of x and their ancestors, as the restaurants at all other nodes are empty. The explicitly-stored tree size is bounded by the product of the number of distinct x_i and the height of the tree.

However, the branch-and-bound algorithm to find the most probable source node x needs to be adapted, since the `branch` function would otherwise return an infinite number of subsets. Consider any node n that has no observations (i.e. has an empty restaurant). Then by equation 50, $p(\ ^{\mathbf{n}}_{\text{new}}) = p(\ ^{\text{parent}(\mathbf{n})}_{\text{new}}) = \cdots = p(\ ^{\mathbf{a}}_{\text{new}})$ where $\mathbf{a}$ is the most recent non-empty ancestor of $\mathbf{n}$. For such nodes, the objective function in equation 64 can be simplified

$$\frac{p(\mathbf{n})}{N_{\text{samples}}} \sum_{\mathbf{z}^{(t)}\ \mathbf{z}|\mathbf{x},\mathbf{y}} p(\ ^{\mathbf{a}}_{\text{new}} = y\ ). \tag{73}$$

Aside from the prior term $p(\mathbf{n})$, all empty descendant nodes of $\mathbf{a}$ have the same objective function value, which is independent of $\mathbf{n}$. So to adapt the algorithm to the infinite hierarchy case, the `branch` function is modified:

$$\texttt{branch}(D(\mathbf{a})) \textit{ returns } \left(\{\mathbf{a}\}, D(\mathbf{c}_1), \cdots, D(\mathbf{c}_n), \bigcup_{i=n+1}^{1} D(\mathbf{c}_i)\right), \tag{74}$$

where $(\mathbf{c}_1, \cdots, \mathbf{c}_n)$ are the non-empty child nodes of $\mathbf{a}$, and $(\mathbf{c}_{n+1}, \mathbf{c}_{n+2}, \cdots)$ are the empty child nodes of $\mathbf{a}$. Next, in algorithm 8, following line 8, we add a new else-if statement to check for the case that S is a set of empty nodes. If so, S is added to C, and we don't continue the search in the empty descendant nodes. The resulting adapted branch-and-bound algorithm correctly and efficiently solves the optimization problem for infinite hierarchies.

4.1.4 *Modeling dependence on discrete structures*

HDPs can be used to learn distributions that depend on sequences of non-negative integers. Consider the data $\{(x_1, y_1), \cdots, (x_n, y_n)\}$ where each $x_i \in \mathbb{Z}_+^h$ is a sequence of h non-negative integers . The distribution of y_i is dependent on the value of x_i. We can use the HDP to learn the relationship of this dependence: construct a hierarchy of height h, where each non-leaf node has a countably infinite number of children, every child node corresponding to a non-negative integer. Here, each x_i uniquely identifies a leaf node in the hierarchy by characterizing a path from the root $\mathbf{o}$ to a leaf: the first integer in the sequence identifies

the child of the root node, the second integer identifies the grandchild, and so on. The y_i are then sampled from the corresponding leaf node. We can apply the MCMC algorithms derived above to learn the distributions of the y_i, and how those distributions relate to the integer sequences x_i.

Given a new observation y, the branch-and-bound algorithm can be used to find the most probable corresponding integer sequence x, but we need to be able to convert the output of the branch-and-bound into the corresponding integer sequence. The algorithm will output a list of the k most probable source nodes from which y is sampled, *or* sets of empty source nodes (since the HDP hierarchy is infinite). More precisely, let $(o_1, \ldots, o_k)$ be the output of the branch-and-bound algorithm. For each o_j, there are two cases:

1. o_j is a single leaf node, in which case it is straightforward to convert the node into its corresponding integer sequence.

2. o_j is the set of empty descendants of a node $\mathbf{a}$. In this latter case, it can be converted into an "incomplete" sequence of integers, where the first $L(\mathbf{a})$ numbers of the sequence correspond to the node $\mathbf{a}$, where $L(\mathbf{a})$ is the level of $\mathbf{a}$. This incomplete sequence represents the set of all integer sequences that begin with the same $L(\mathbf{a})$ integers, that do not already explicitly exist in the tree.

For example, let $\mathbf{n}_a$ be the a^{th} child of the root node $\mathbf{o}$ in the HDP hierarchy. Let $\mathbf{n}_{a,b}$ be the b^{th} child of $\mathbf{n}_a$, and so on. Suppose the training set contains only the sequences $(4,3,1)$, $(4,7,4)$, and $(4,8,2)$. Therefore, the nodes in the HDP with non-empty restaurants are: $\mathbf{n}_{4,3,1}$, $\mathbf{n}_{4,7,4}$, $\mathbf{n}_{4,8,2}$, $\mathbf{n}_{4,3}$, $\mathbf{n}_{4,7}$, $\mathbf{n}_{4,8}$, $\mathbf{n}_4$, and $\mathbf{o}$. If the branch-and-bound algorithm returns $\mathbf{n}_{4,7,4}$, the corresponding output integer sequence is $(4,7,4)$. If instead, branch-and-bound returns the set of the empty descendant nodes of $\mathbf{n}_4$, the corresponding output integer sequence is $(4, \ \backslash\{3,7,8\}, \ )$. The ' ' is a "wildcard" symbol that represents the set of all non-negative integers. Thus, $(4, \ \backslash\{3,7,8\}, \ )$ represents the set of all integer sequences that start with $(4,\ldots)$ but do not start with $(4,3,\ldots)$, $(4,7,\ldots)$, or $(4,8,\ldots)$.

This model can be extended to the case where the $x_i \in X$ have richer structure (e.g. X is the set of labeled trees, graphs, logical forms, etc). To do so, define d functions $f_k : X \to Z_+$ that characterize an aspect of the input structures x_i. We call these functions f_k *feature functions*. For example, if x is a labeled binary tree, $f_1(x)$ returns the label of the root node, and $f_2(x)$ returns the label of the left child, etc. The functions serve to map the structures x_i into sequences of non-negative integers: $(f_1(x_i), \ldots, f_d(x_i))$. Then the above HDP model can be directly used to learn the relationship between these integer sequences and the distribution of the observations y_i. For a new observation y, the branch-and-bound algorithm will return the k most likely integer sequences (possibly with wildcard symbols) that

represent the unknown structure x . To convert the integer sequence $(w_1, :::, w_d)$ into the corresponding structure in X, we can compute:

$$f_1^{-1}(w_1) \bigcap ::: \bigcap f_d^{-1}(w_d) \text{ where } f_k^{-1}(w_k) \ , \ fx : f_k(x) \ 2 \ w_k g. \quad (75)$$

PWL implements three functions to perform the above mapping between integer sequences and more structured representations in X:

1. `get_feature(f, X)`: Given a feature function f and a set X X, return ff(x) : x 2 Xg.
2. `set_feature(f, X^old, w)`: Given a feature function f, a set X^{old} X, and a non-negative integer w 2 Z_+, return $X^{old} \setminus f^{-1}(w)$. This function is used in the case that w_k is an integer (not a wildcard).
3. `exclude_features(f, X^old, W)`: Given a feature function f, a set X^{old} X, and a finite set of non-negative integers W 2 Z_+, return $X^{old} \cap f^{-1}(W)$. This is used in the case that w_k is a wildcard n W.

```
S  !  N:select_arg1 VP:delete_arg1
VP !  V:identity N:select_arg2
VP !  V:identity
N  !  "New Jersey"        V  !  "borders"
N  !  "NJ"                V  !  "bordered"
N  !  "Pennsylvania"      V  !  "has"
N  !  "Michael Phelps"    V  !  "swims"
N  !  "tennis"            V  !  "plays"
```

Figure 10: Example of a grammar in our framework. This example grammar operates on logical forms of the form *predicate(first argument, second argument)*. The semantic function `select_arg1` returns the first argument of the logical form. Likewise, the function `select_arg2` returns the second argument. The function `delete_arg1` removes the first argument, and `identity` returns the logical form with no change. In our work, the interior production rules (the first three listed above) are examples of rules that we specify, whereas the terminal rules and the posterior probabilities of *all* rules are learned via grammar induction. A simplified semantic representation is shown here for the sake of illustration. PWM uses a richer semantic representation. Section 4.2.2 provides more detail.

4.2 MODEL: SEMANTIC GRAMMAR

A grammar in our formalism operates over a set of nonterminals N and a set of terminal symbols W. It can be understood as an extension of

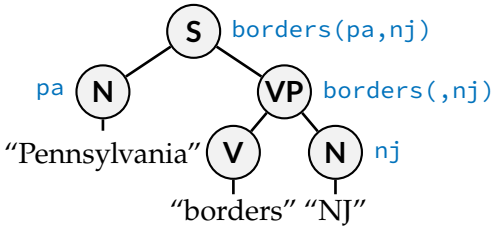


Figure 11: Example of a derivation tree under the grammar given in Figure 10. The logical form corresponding to every node is shown in blue beside the respective node. The logical form for V is `borders(,nj)` and is omitted to reduce clutter.

a context-free grammar (CFG) (Chomsky, 1956) where the generative process for the syntax is dependent on a logical form, thereby coupling syntax with semantics. In the top-down generative process of a derivation tree, a logical form guides the selection of production rules. Production rules in our grammar have the form $A \to B_1 : f_1 ::: B_k : f_k$ where $A \in N$ is a nonterminal, $B_i \in N \cup W$ are right-hand side symbols, and f_i are *semantic transformation functions*. These functions describe how to "decompose" this logical form when recursively generating the subtrees rooted at each B_i. Thus, they enable semantic compositionality. An example of a grammar in this framework is shown in Figure 10, and a derivation tree is shown in Figure 11. Let R be the set of production rules in the grammar and R_A be the set of production rules with left-hand nonterminal symbol A.

4.2.1 *Generative process*

A *derivation tree* in this formalism is a tree where every interior node is labeled with a nonterminal symbol in N, every leaf is labeled with a terminal in W, and the root node is labeled with the root nonterminal S. Moreover, every node in the tree is associated with a logical form: let $x^{\mathbf{n}}$ be the logical form assigned to the tree node $\mathbf{n}$, and $x^{\mathbf{o}} = x$ for the root node $\mathbf{o}$.

The generative process to build a derivation tree begins with the root nonterminal S and a logical form x. Recall that in PWM, the logical form x is the conclusion of each proof generated from the theory, as described in section 3.2.2, but other prior distributions may be used for x, and the presentation here will be agnostic to the choice of this prior. PWM *expands* S by randomly drawing a production rule from R_S, *conditioned* on the logical form x. This provides the first level of child nodes in the derivation tree. For example, if the rule $S \to B_1 : f_1 ::: B_k : f_k$ were drawn, the root node would have k child nodes, $\mathbf{n}_1, :::, \mathbf{n}_k$, respectively labeled with the symbols $B_1, :::, B_k$. The logical form associated with each node is determined by the semantic transformation function: $x^{\mathbf{n}_i} = f_i(x^{\mathbf{o}})$. These functions describe the relationship between the logical form at a child node and that of its

parent node. This process repeats recursively with every right-hand side nonterminal symbol, until there are no unexpanded nonterminal nodes. The sentence is obtained by taking the *yield* (i.e. the concatenation) of the terminals in the tree.

The semantic transformation functions are specific to the semantic formalism and may be defined as appropriate to the application. In our semantic parsing experiments in section 4.4, we define a domain-independent set of transformation functions specific to the Datalog representation of GeoQuery and Jobs (e.g., one function selects the left n conjuncts in a conjunction, another selects the n^{th} argument of a predicate instance, etc). Some examples of these transformation functions are:

- The function `select_left` returns the left conjunct of a conjunction. For example, given the Datalog expression `(river(A), loc(A,B),const(B,stateid(colorado)))`, this function returns `river(A)`.
- The function `delete_left` returns a conjunction where the first conjunct is removed. For example, given `(river(A),loc(A,B), const(B,stateid(colorado)))`, this function returns `(loc(A,B), const(B,stateid(colorado)))`.
- The function `select_arg2` returns the second argument in an atomic formula. For example, given `const(A,stateid(maine))`, this function returns `stateid(maine)`.

In PWL, we define a different set of domain-independent transformation functions for a new semantic formalism based on higher-order logic that we will present in section 4.5.

Semantic transformation functions are allowed to fail, which is useful in defining richer transformation functions and providing more flexibility when designing the production rules of the grammar. If in the generative process, a transformation function returns failure, the generative process is restarted from the root (all progress up to the failure is discarded). As an example, failure enables the definition of transformation functions that check whether the input logical form satisfies a specific property: `require_binary_conjunction` returns the input logical form, unchanged, if it is a conjunction of length 2; otherwise, it returns failure. Since failure can cause the generative process to repeatedly restart, the process of sampling using the generative process can be expensive. However, PWL does not generate sentences using this algorithm , and as we show in section 4.3, the performance of PWL is not adversely affected.

4.2.2 *Selecting production rules*

The above description does not specify the conditional distribution from which rules are selected from R_A given the logical form. There

are many modeling options available in choosing this distribution, but we need a distribution that captures complex dependencies between the logical form and selected production rule. For example, consider the grammar in figure 10 and the logical form `plays_sport(michael_phelps,tennis)`. When generating a sentence for this logical form, at the root nonterminal S, there is only one production rule available, S ! N:`select_arg1` VP:`delete_arg1`, so this rule is selected. Now consider the child node corresponding to the nonterminal VP. Its logical form is `plays_sport(,tennis)`, which is the output of the function `delete_arg1` when applied to the logical form at the root node. At this point, there are two choices of production rules with VP on the left-hand side: VP ! V N and VP ! V. In this case, we want the conditional distribution to select VP ! V N, since the most likely sentence that conveys the semantics in the logical form is "plays tennis." In fact, VP ! V N should be selected even in when the logical form is `plays_sport(,baseball)` or `plays_sport(,soccer)` or almost any other sport. However, if the logical form were `plays_sport(,swimming)`, we want the conditional distribution to give higher probability to VP ! V, since the verb phrase "swims" is much more likely. Therefore, a desirable property of the conditional distribution for VP production rules is that the distribution depends primarily on the predicate symbol (e.g. `plays_sport`) but also depends secondarily on the object argument (e.g. `swimming` or `tennis`).

PWL uses the HDP model, as presented in section 4.1.4, to capture this dependence. Every nonterminal in our grammar A 2 N will be associated with an HDP hierarchy. For each nonterminal, we specify a sequence of *semantic feature functions*, $\mathsf{fg}_1, \ldots, \mathsf{g}_m\mathsf{g}$, each of which, when given input logical form x, returns a non-negative integer. The HDP hierarchy is a complete infinite tree of height m, where every parent node has an infinite number of child nodes, one for each non-negative integer. The base distribution H at the root of the HDP is over R_A.

Take, for example, the derivation in Figure 11. In the generative process where the node VP is expanded, the production rule is drawn from the HDP associated with the nonterminal VP. Suppose the HDP was constructed using a sequence of two semantic features: (`predicate`, `arg2`). In the example, the feature functions are evaluated with the logical form `borders(,nj)` and they return a sequence of two integers, the first is the identifier for the symbol `borders` and the second is the identifier for the symbol `nj`. This sequence uniquely identifies a path in the HDP hierarchy from the root node **o** to a leaf node **n**. The production rule VP ! V N is drawn from this leaf node $\mathsf{G}^{\mathbf{n}}$, and the generative process continues recursively. As desired, the distribution of the selected production rule $\mathsf{G}^{\mathbf{n}}$ depends on the HDP source node **n**, which itself depends primarily on the first fea-

ture and secondarily on the second feature and so on (in this example, the `predicate` and `arg2` of the logical form are the first and second features, respectively).

In our implementation, the set of nonterminals N is divided into two disjoint groups: (1) the set of "interior" nonterminals, and (2) preterminals. The production rules of preterminals are restricted such that the right-hand side contains only terminal symbols. The rules of interior nonterminals are restricted such that only nonterminal symbols appear on the right side.

1. For **preterminals**, H is a distribution over sequences of terminal symbols. The sequence of terminal symbols is distributed as follows: first sample the length of the terminal from a geometric distribution (i.e. the number of words) and then generate each word in the sequence i.i.d. from a uniform distribution over a finite set of (initially unknown) terminals. Note that we do not specify a set of domain-specific terminal symbols in defining this distribution.

2. For **interior nonterminals**, H is a discrete distribution over a domain-independent set of production rules, which we specify. Since the production rules contain transformation functions, they are specific to the semantic formalism. However, prior knowledge of the English language can be encoded in these specified production rules, which dramatically improves the statistical efficiency of our model and obviates the need for massive training sets to learn English syntax. It is nonetheless tedious to design these rules while maintaining domain-generality. Once specified, however, these rules in principle can be re-used in new tasks and domains without further changes.

We emphasize that only the prior is specified here, and PWL uses grammar induction to infer the posterior. In principle, a more relaxed choice of H may enable grammar induction without pre-specified production rules, and therefore without dependence on a particular semantic formalism or natural language, if an efficient inference algorithm can be developed in such cases.

4.2.3 *Modeling morphology*

The grammar model is easily modified to incorporate word morphology. To do so, we add an additional step to the generative process after generating the terminal symbols. Instead of the terminal symbols constituting the tokens of the sentence directly, the terminal symbols are instead word roots coupled with morphological flags that indicate their inflection. For example, in the grammar in figure 10, rather than having multiple rules for the various inflections of the verb "to border", such as V *!* "borders", V *!* "bordered", V *!* "bordering", there would

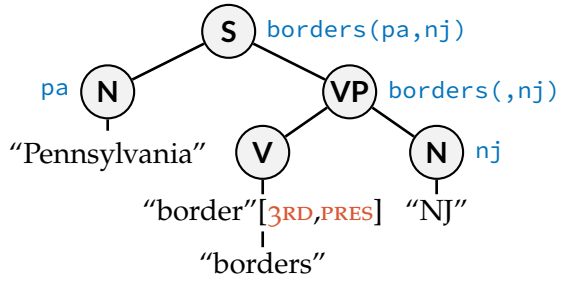


Figure 12: Example of a derivation tree under a grammar with a model of morphology. The logical form corresponding to every node is shown in blue beside the respective node. The logical form for V is `borders(,nj)`[3RD,PRES] and is omitted to reduce clutter. Morphology is not modeled for proper nouns such as "Pennsylvania" and "NJ."

only be a single production rule for the root: V ! "border". In order to produce the various inflections, the logical form is augmented to carry morphology information. Semantic transformation functions may add or modify morphological flags. For example, suppose we have the rule VP ! V:`add_third_person,add_present_tense` where `add_third_person` is a function that adds the 3RD flag (indicating third person) to the logical form, and `add_present_tense` is a function that adds the PRES flag (indicating present tense) to the logical form. These morphological flags are copied into the terminal symbols, and as a final step, the roots are inflected according to the morphological flags (e.g. "border[3RD,PRES]" is inflected to "borders"). See figure 12 for an example of a derivation tree for a grammar that models morphology.

If a root with morphological flags has multiple inflections, such as "octopus"[PL] (PL indicates plural), the generative process selects one uniformly at random. During inference (i.e. parsing), this morphological model has the effect of performing morphological and syntactic-semantic parsing jointly, as we will show in the next section. Wiktionary (Wikimedia Foundation, 2020) provides comprehensive high-quality morphology information for English verbs, common nouns, adjectives, and adverbs. PWL uses Wiktionary to construct a mapping between uninflected roots and inflected words, which is used in both directions: (1) given root and morphological flags, find the corresponding set of inflections, or (2) given an inflected word, find the corresponding set of roots and morphological flags. Note that only (2) is necessary for parsing and training, whereas (1) is necessary for generation.

Although this morphology model is implemented in PWL, we do not use it in our experiments on GeoQuery and Jobs. The morphology model is used in the experiments described later in the thesis.

4.3 INFERENCE AND IMPLEMENTATION

4.3.1 *Training*

In this section, we describe how to infer the latent derivation trees $t = \{t_1, \ldots, t_n\}$, given a collection of sentences $y = \{y_1, \ldots, y_n\}$ and logical form labels $x = \{x_1, \ldots, x_n\}$, where each derivation tree t_i is distributed according to the conditional distribution described by the generative process in section 4.2.1 above.

We describe grammar induction independently of the choice of rule distribution. We wish to compute the posterior $p(t \mid x, y)$ of the latent derivation trees. This is intractable to compute exactly, and so we resort to MCMC. To perform blocked Gibbs sampling, we pick initial values for t and repeat the following: For $i = 1, \ldots, n$, sample $t_i \mid t_{-i}, x_i, y_i$ where $t_{-i} = t \setminus \{t_i\}$.

$$p(t_i \mid t_{-i}, x_i, y_i) \propto \mathbb{1}\{\text{yield}(t_i) = y_i\} \prod_{A \in N} p\left(\bigcap_{\substack{\mathbf{n} \in t_i : \mathbf{n} \\ \text{has label } A}} r^{\mathbf{n}} \;\middle|\; t_{-i}, x_i \right), \quad (76)$$

where N is the set of nonterminals, and the intersection is taken over the nodes $\mathbf{n} \in t_i$ labeled with the nonterminal A in the i^{th} derivation tree, and $r^{\mathbf{n}}$ is the production rule at node $\mathbf{n}$. Note that the probability does not necessarily factorize over rules, as is the case when using the HDP. So in order to sample t_i, we use Metropolis-Hastings (MH), where the proposal distribution is given by the fully factorized form:

$$p(t_i \mid t_{-i}, x_i, y_i) \propto \mathbb{1}\{\text{yield}(t_i) = y_i\} \prod_{\mathbf{n} \in t_i} p(r^{\mathbf{n}} \mid t_{-i}, x_i^{\mathbf{n}}). \quad (77)$$

The algorithm for sampling t_i is detailed in section 4.3.1.1. After sampling t_i', we choose to accept the new sample with probability

$$\min \left\{ 1, \frac{\prod_{\mathbf{n} \in t_i'} p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, t_{-i})}{p\left(\bigcap_{\mathbf{n} \in t_i'} r^{\mathbf{n}} \mid x, t_{-i}\right)} \frac{p\left(\bigcap_{\mathbf{n} \in t_i} r^{\mathbf{n}} \mid x, t_{-i}\right)}{\prod_{\mathbf{n} \in t_i} p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, t_{-i})} \right\}, \quad (78)$$

where t_i, here, is the old sample, and t_i' is the newly proposed sample. In practice, this acceptance probability is very high. This approach is very similar in structure to that in Blunsom and Cohn (2010), Cohn, Blunsom, and Goldwater (2010), and Johnson, Griffiths, and Goldwater (2007).

Computing the conditional probabilities of the production rules $p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, t_{-i})$ and $p(\bigcap_{\mathbf{n} \in t_i'} r^{\mathbf{n}} \mid x, t_{-i})$ (as well as the quantities required in sampling t_i') depends on the model for selecting production rules. In PWL, which uses an HDP model, these quantities can be computed using equations 52 and 55. PWL only keeps the last MCMC sample ($N_{\text{samples}} = 1$), so for each node in every derivation tree $\mathbf{m} \in t_j$, the production rule at that node $r^{\mathbf{m}}$ corresponds to a customer in the

Chinese restaurant representation of the HDP associated with the nonterminal at node **m**. When resampling the derivation tree t_i, PWL removes all customers that correspond to a production rule in t_i. Then it is straightforward to compute conditional probabilities according to equations 52 and 55 with the remaining customers. Once a new t_i is sampled, the customers corresponding to production rules in t_i are added to their respective restaurants.

There may be additional random variables in the grammar apart from the derivation trees, such as in the HDPs. We perform Gibbs sampling steps for these variables after each loop of resampling the trees $t_i, i = 1, \ldots, n$. The grammar induction algorithm is summarized: Pick initial values for t and and repeat the following,

1. For $i = 1, \ldots, n$, sample t_i j $, t_{-i}, x_i, y_i$ from the distribution given by equation 77. Then accept this sample as the new value for t_i with probability given by equation 78.

2. Perform the Gibbs sampling step for j t.

In all experiments throughout the thesis, we run the above loop for 10 iterations. Note that this algorithm requires no further supervision beyond the utterances y and logical forms x. However, it is able to exploit additional information such as supervised derivation trees: if $\bar{t}$ t is a subset of derivation trees that are supervised, the Gibbs sampling algorithm simply avoids resampling the trees in $\bar{t}$. These supervised derivation trees do not necessarily need to be rooted in the nonterminal S. For example, a lexicon can be provided where each entry is a terminal symbol y_i with a corresponding logical form label x_i. In our experiments on GeoQuery and Jobs, we evaluate our method with and without such a lexicon.

4.3.1.1 *Sampling* t_i

To sample from equation 77, we use *inside-outside sampling* (Finkel, Manning, and Ng, 2006; Johnson, Griffiths, and Goldwater, 2007), a dynamic programming approach. For every nonterminal A *2* N, sentence start position i, end position j, and logical form x, let $I_{(A,i,j,x)}$ be the probability that t_i has a node **n** with the label A and logical form x and spans the sentence from i to j. This is known as the *inside probability*. Similarly, for all production rules in the grammar A *!* $B_1{:}f_1 ::: B_K{:}f_K$, sentence boundary positions between the right-hand side nonterminals $l_1 < ::: < l_{K+1}$, and logical forms x, let $I_{(S\ !\ B_1:f_1:::B_K:f_K,l,x)}$ be the probability that t_i has a node **n** with the label A and logical form x and has child nodes B_u, each with logical forms $f_u(x)$, and each spanning the sentence from l_u to l_{u+1}. This is known as the *inside rule probability*. Note that we don't need to compute all possible inside probabilities for all logical forms (in many applications, the set of logical forms is infinite). Therefore, we compute

these inside probabilities top-down, beginning at the root nonterminal $\beta_{(S,0,|y_i|,x_i)}$ with the known logical form x_i where $|y_i|$ is the length of sentence y_i. The following formula can be used to compute this quantity recursively:

$$\beta_{(A,i,j,x)} = \sum_{A \to B_1:f_1 \cdots B_K:f_K} \sum_{i=l_1<\cdots<l_{K+1}=j} \beta_{(A \to B_1:f_1 \cdots B_K:f_K, \mathbf{l}, x)}. \quad (79)$$

$$\beta_{(A \to B_1:f_1 \cdots B_K:f_K, \mathbf{l}, x)} = p(A \to B_1:f_1 \cdots B_K:f_K \mid x, t_{-i}) \prod_{u=1}^{K} \beta_{(B_u, l_u, l_{u+1}, f_u(x))}. \quad (80)$$

If $f_u(x)$ returns failure, then $\beta_{(B_u,l_u,l_{u+1},f_u(x))} = 0$. Note that in the case that A is a preterminal,

$$\beta_{(A \to w, l_1, l_2, x)} = 1\{w \text{ matches } y_i \text{ at } (l_1, l_2)\}\, p(A \to w \mid t_{-i}). \quad (81)$$

where w is a terminal. Aside from the inside probabilities that were required to compute the root inside probability $\beta_{(S,0,|y_i|,x_i)}$, all other inside probabilities are 0. In PWL, this recursion is implemented iteratively, in order to avoid any issues with limited stack size and to share code with the parsing and generation algorithms. We also take care not to recompute previously computed inside probabilities.

All that remains is the outside step: sample the derivation tree using the computed inside probabilities. To do so, start with the root nonterminal S at positions $i = 0$ to $j = |y_i|$ and logical form x_i, and consider all production rules with S on the left-hand side $S \to B_1:f_1 \cdots B_K:f_K$ and all sentence boundaries between the right-hand side nonterminals $l_1 < \cdots < l_K < l_{K+1}$ where $l_1 = i$ and $l_{K+1} = j$. Sample a production rule and sentence boundaries with probability proportional to the inside rule probability $\beta_{(S \to B_1:f_1 \cdots B_K:f_K, \mathbf{l}, x_i)}$. Next, consider each right-hand side nonterminal of the selected rule B_u, start position l_u, end position l_{u+1}, and logical form $f_u(x_i)$, and recursively repeat the sampling procedure. The end result is a tree sampled from equation 77.

4.3.2 *Parsing*

For a new sentence y', we aim to find the logical form x' and derivation t' that maximizes

$$p(x', t' \mid y', x, y) = \int p(x', t' \mid y', t)\, p(t \mid x, y)\, dt, \quad (82)$$

$$\frac{1}{N_{\text{samples}}} \sum_{y', t \mid x, y} p(x', t' \mid y', t). \quad (83)$$

These samples of t are obtained from the above training procedure. For the parsing approach presented in this section, it is assumed that

$N_{\text{samples}} = 1$, and so $p(x, t \mid y, x, y) \approx p(x, t \mid y, t)$, where t is the last MH sample from the training procedure. Thus, we can write the objective function for parsing:

$$p(x, t \mid y, t) \propto p(x)p(y \mid t)p(t \mid x, t),$$

$$= 1\{\text{yield}(t) = y\}\, g p(x) p\left(\bigcap_{n \in t} r^n \;\middle|\; x^n, t\right), \quad (84)$$

$$1\{\text{yield}(t) = y\}\, g p(x) \prod_{n \in t} p(r^n \mid x^n, t).^1 \quad (85)$$

This is a discrete optimization problem, which we solve using *branch-and-bound* (see algorithm 8). The algorithm starts by considering the set of all derivation trees of y and partitions it into a number of subsets (the "branch" step). For each subset S, we compute an upper bound on the log probability of any derivation in S (the "bound" step). This bound is given by equations 87, 88, and 89. Having the computed the bound for each subset, we push them onto a priority queue, prioritized by the bound. We then pop the subset with the highest bound and repeat this process, further subdividing this set into subsets, computing the bound for each subset, and pushing them onto the queue. Eventually, we will pop a subset containing a single derivation which is provably optimal, if its objective function value according to the above equation is at least the priority of the next item in the queue. We can continue the algorithm to obtain the top-k derivations/logical forms. Since this algorithm operates over *sets* of logical forms (where each set is possibly infinite), we must implement a data structure to sparsely represent such sets of formulas, as well as algorithms to perform set operations, such as intersection and subtraction.

Each set of derivations is sparsely represented in PWL as a single incomplete derivation tree (i.e. the leaf nodes may be either terminals or nonterminals) and a logical form set. The logical form set represents the logical form of the root node of every derivation tree in the set. The logical forms at the other nodes can be computed by using the semantic transformation functions. In addition, every nonterminal node with non-zero children has two integer indices that indicate its start and end positions in the sentence y. This data structure represents the set of all derivation trees whose nodes match the nodes in the incomplete derivation tree at the given sentence positions. In addition, each derivation tree set has an integer counter to indicate to the `branch` function how to subdivide the set. Each such set of derivation trees is also called a *search state*. As an example, consider the input sentence "Trenton is the capital of New Jersey." Now consider a search state where the incomplete derivation tree contains only a single node

1 Equations 84 and 85 are not equal since the conditional distribution of production rules is not necessarily i.i.d. PWL uses an HDP for this conditional distribution, which has the nice property that as the size of the training set increases, this approximation becomes more exact.

labeled NP with start position 3 and end position 7 (corresponding to "capital of New Jersey") and the logical form set is the set of all logical forms. This search state represents the set of all derivation trees that have a node with label NP and is the common ancestor of the terminals in "capital of New Jersey."

Given a set of derivation trees, the `branch` function is defined in algorithm 9. The branch-and-bound algorithm is started with a derivation tree set whose incomplete derivation tree has a single root node with nonterminal S, the set of all logical forms, start position 0, and end position $|y|$.

There are two missing pieces in algorithm 9: the first is on line 17. To compute this, we can augment the branch-and-bound algorithm to return the m^{th} best element(s) that maximize(s) an objective function over a set. This augmented function is shown in algorithm 11. Whenever line 17 is first executed for a given nonterminal B_k, start position i_k, end position j_k and logical form set $f_k(X)$, initialize the priority queue Q in algorithm 11 with: $Q.\text{push}(S', h(S'))$ where S' is the search state with an incomplete derivation tree consisting of a single node at the root with nonterminal B_k, start position i_k, end position j_k, and logical form set $f_k(X)$.

The other missing piece in algorithm 9 is line 28, which depends on the model for selecting production rules. PWL uses an HDP model, and is able to directly use the algorithm described in section 4.1.2 to compute X. Algorithm 11 may also be used here to return the m^{th} most likely logical form(s).

The above branch-and-bound algorithm requires a heuristic function that, for an input search state (set of derivation trees), returns an upper bound on the objective function in equation 85 over all derivation trees in the set. This heuristic function determines the order of the search states to visit. The product in the objective $\prod_{n \in t} p(r^n \mid x^n, t)$ can be decomposed accordingly into a product of two components: (1) the *inner probability* at a node $\mathbf{n} \in t$ is the product of the terms that correspond to the subtree rooted at $\mathbf{n}$, and (2) the *outer probability* is the product of the remaining terms, which correspond to the parts of t outside of the subtree rooted at $\mathbf{n}$.

To help define this heuristic, we define an upper bound on the log inner probability $I_{(A,i,j)}$ for any derivation tree rooted at nonterminal A at start position i and end position j in the sentence.

$$I_{(A,i,j)} \geq \max_{A \to B_1 \cdots B_K} \Bigg(\max_{x'} \log p(A \to B_1, \ldots, B_K \mid x', t) + \max_{l_2 < \cdots < l_K} \sum_{k=1}^{K} I_{(B_k, l_k, l_{k+1})} \Bigg), \quad (86)$$

where $l_1 = i$, $l_{K+1} = j$. Note that the left term is a maximum over all logical forms x', and so this upper bound only considers syntactic information. Computing the left term depends on the model for

Algorithm 9: Pseudocode for `branch` in the branch-and-bound algorithm for the parser, which aims to maximize equation 85.

```
function branch(derivation tree set S)
    L is an empty list
    n is the root of the incomplete derivation tree of S
    X is the logical form set at n
    i is the start sentence position of n
    j is the end sentence position of n
    if n has no child nodes
        A is the nonterminal symbol of n
        return expand(A, i, j, X) /* see algorithm 10 */
    else if n has a nonterminal child node with no children
        A ! B_1:f_1 ::: B_K:f_K is the production rule at n
        c_k is the first nonterminal child node of n with no children
        B_k is the nonterminal symbol of c_k
        i_k is the start sentence position of c_k
        j_k is the end sentence position of c_k
        m is the counter of S
        S_m is the m^th most probable set of derivation trees with root nonterminal
          B_k, start position i_k, end position j_k, whose logical forms are a subset of
          f_k(X) , ff_k(x) 6= fail : x 2 X_g, according to equation 85
        if S_m exists
            for sentence positions j_{k+1} such that j_k < j_{k+1} < j do
                /* the operation X \ f_k^{-1}(X_m) can return a union of sets */
                let X_{k,1} [ ::: [ X_{k,r} be the output of X \ f_k^{-1}(X_m) where X_m is the logical
                  form set of S_m, and f_k^{-1}(X_m) , fx : f_k(x) 2 X_m g
                for X_{k,l} 2 fX_{k,1}, ::: , X_{k,r} g do
                    S is a new derivation tree set with counter 1, the incomplete
                      derivation tree is identical to that of S except c_k is substituted with
                      the incomplete derivation tree of S_m, the logical form set at the root is
                      X_{k,l}, and the end position of c_{k+1} is j_{k+1}
                    L.add(S )
            L.add( a new derivation tree set identical to S except its counter is m + 1)
    else
        A ! B_1:f_1 ::: B_K:f_K is the production rule at n
        m is the counter of S
        X_m is the m^th most likely set of logical forms according to
          p(A ! B_1:f_1 ::: B_K:f_K j x 2 X, t)
        if X_m exists
            L.add( a new derivation tree set identical to S except its logical form set is
              X_m, and is marked as COMPLETE )
            L.add( a new derivation tree set identical to S except its counter is m + 1)
    return L
```

Algorithm 10: Pseudocode for the `expand` helper function, which algorithm 9 invokes.

```
function expand(nonterminal A, start position i, end position j, logical form set X)
    L is an empty list
    if A is a preterminal
        for rules A → w where the tokens in the sentence y matches the terminal w at
          positions i to j do
            /* if using the morphology model, we instead require that w
               is a valid morphological parse of the tokens in the
               sentence y at positions i to j */
            S is a new derivation tree set where the incomplete derivation tree
              consists of a root node n with nonterminal A, start position i, end
              position j, logical form set X, and child node w
            L.add(S)
    else
        for rules A → B_1:f_1 ::: B_K:f_K and sentence positions k such that i < k < j do
            S is a new derivation tree set with counter 1, the incomplete derivation
              tree consists of a root node n with nonterminal A, start position i, end
              position j, logical form set X, and for each child node c_i, the nonterminal
              is B_i, and logical form set is f_i(X) , f f_i(x) g fail : x 2 Xg; the start position
              of c_1 is i, the end position of c_1 is k, and the end position of c_K is j
            L.add(S)
    return L
```

Algorithm 11: A modified branch-and-bound algorithm to return the k^{th} best element(s) that maximize(s) the function f. Before the first call to this function, C is initialized as an empty list, and Q is initialized with a single element: `Q.push(X, h(X))` where X is the domain on which to maximize f. The changes to C and Q persist across subsequent calls to `get_kth_best`.

```
function get_kth_best(objective function f,
                      heuristic h,
                      priority queue Q,
                      list of completed elements C,
                      integer k)
    if there are at least k elements in C with priority at least the highest priority in Q
        return k-best element in C
    while Q not empty do
        (S, v) = Q.pop()
        if S is marked as COMPLETE
            C.add(x, f(x))
        else
            (S_1, ::: , S_n) = branch(S)
            for i = 1, ::: , n do
                Q.push(S_i, h(S_i))
        /* check termination condition */
        if there are at least k elements in C with priority at least v
            return k-best element in C
    return ?
```

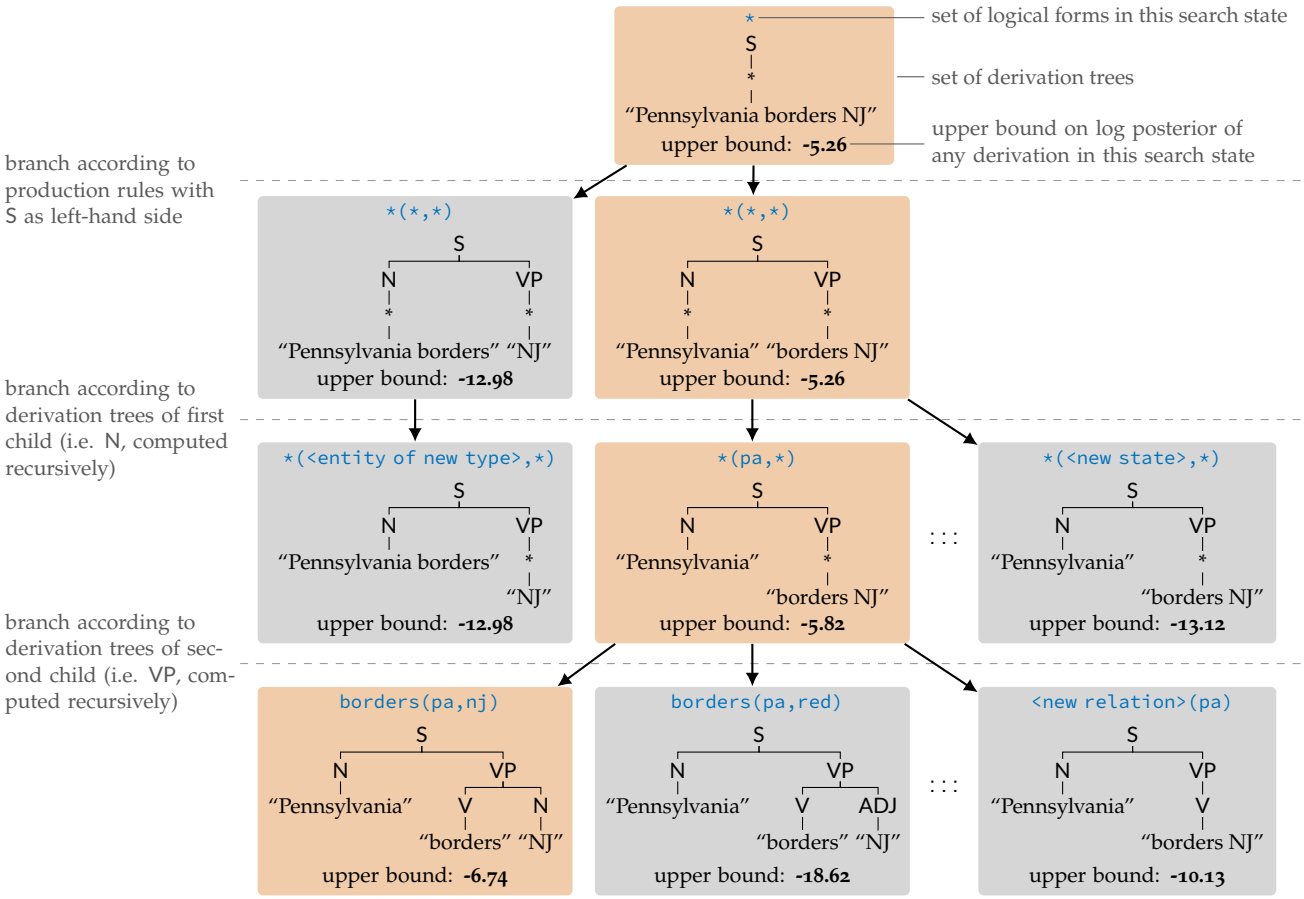


Figure 13: The search tree of the branch-and-bound algorithm during parsing. In this diagram, each block is a search state, which represents a set of derivation trees. The blue asterisk * denotes the set of all possible logical forms, whereas the black asterisk * denotes the set of all possible derivation (sub)trees. Note only the logical form at the root node is shown. The gray-colored search states are unvisited by the parser, since their upper bounds on the log posterior are smaller than that of the completed parse at the bottom of the diagram (-6.74), thus allowing the parser to ignore a very large number of improbable logical forms and derivations. In this example, we use the grammar from figure 10. The branching steps here are simplified for the sake of illustration. The recursive optimization of the derivation subtrees for N and VP are not shown, which have their own respective search trees.

selecting production rules. Since PWL uses an HDP, it uses the branch-and-bound approach in section 4.1.2 to compute this term. The right term can be maximized using dynamic programming with running time $O(K^2)$. As such, classical syntactic parsing algorithms can be applied to compute I for every chart cell in $O(n^3)$. For any terminal symbol w *2* W, we define $I_{(w,i,j)} = 0$.

We now define the upper bound heuristic on any search state S with an incomplete derivation tree that has root node **n**, start position i, end position j, and logical form set X. If **n** has no child nodes:

$$\log h(S) \;,\; h^{\mathbf{n}}_{x}(X) + l_{(A,i,j)}. \tag{87}$$

Else, if **n** has a nonterminal child node without children, the production rule at **n** is $A \;!\; B_1{:}f_1 \,:\!:\!: B_K{:}f_K$, k is the smallest index of a nonterminal child node, and m is the value of the counter:

$$\log h(S) \;,\; \min\{h^{\mathbf{n}}_{x}(X) + l_{(B_k,l_k,l_{k+1})}, \log \quad_{m-1}\} \\ + \quad + \max_{l_{k+2}<:::<l_{K+1}} \sum_{u=k+1} l_{(B_u,l_u,l_{u+1})}. \tag{88}$$

Else, if all the nonterminal child nodes of **n** has children, and m is the value of the counter:

$$\log h(S) \;,\; h^{\mathbf{n}}_{x}(X) + \quad + \log \quad_{m-1}. \tag{89}$$

where

$$, \sum_{\mathbf{m} 2 S \cap \mathbf{n}} \log p(r^{\mathbf{m}} \mid x \; 2 \; X, \mathbf{t}),$$

$h^{\mathbf{n}}_{x}(X) > \max_{x 2 X} \log p(x^{\mathbf{n}})$ is an upper bound on the semantic prior,

$_m$, the objective function value of the m^{th} most probable derivation trees obtained on line 17,

$_m$, m^{th} highest value of $p(A \;!\; B_1{:}f_1 \,:\!:\!: B_K{:}f_K \mid x \; 2 \; X, t)$ obtained on line 28.

The max in the third term of equation 88 can be computed via dynamic programming with running time $O(K^2)$. In the equation for , the sum over $\mathbf{m} \; 2 \; S \cap \mathbf{n}$ is over all nodes in the incomplete derivation tree of S, excluding **n**. To avoid recomputing every time h is invoked, PWL stores it in every search state. Its initial value is 0. In algorithm 9, on line 22, the log probability of the new search state S is equal to the sum of the log probability of the old state S and the log probability of S_m. In line 30, the log probability of the new search state is equal to the sum of the log probability of the old search state S and $\log p(A \;!\; B_1{:}f_1 \,:\!:\!: B_K{:}f_K \mid x \; 2 \; X, t)$. PWL then uses this quantity directly as in the above heuristic. The heuristic also has the nice property that when a search state is marked COMPLETE, its heuristic value is equal to the logarithm of the objective, aside from the prior term. Thus, when computing the objective function, such as checking the termination condition in the branch-and-bound, we only need to compute the prior term.

With a sufficiently tight upper bound on the objective, this algorithm ignores a very large number of subproblems whose upper bound is too high. Figure 13 shows the search tree for the branch-and-bound

algorithm. By ignoring sets of derivation trees with an upper bound smaller than that of the highest-scoring element in the search queue, the parser can ignore a large number of improbable logical forms and derivations. Thus, with a good upper bound, the parser can run in sublinear time with respect to the size of the theory. The parser resembles a generalized version of the Earley parsing algorithm (Earley, 1970).

4.3.3 *Generating sentences*

In contrast with parsing, given a new logical form x, natural language generation is the task of finding the unknown sentence y and derivation tree t. A straightforward way to do this in PWM is to sample $t \mid t, x$, and simply compute $y = \text{yield}(t)$. The sampling follows the generative process directly.

However, in many situations, it is desirable to find the sentence y and derivation t that maximize:

$$p(y, t \mid x, x, y) = \int p(y, t \mid x, t) p(t \mid x, y), \tag{90}$$

$$\frac{1}{N_{\text{samples}}} \sum_{t \; t \mid x, y} p(y, t \mid x, t), \tag{91}$$

$$p(y, t \mid x, t) \quad 1\{\text{yield}(t) = y\} \prod_{n \in t} p(r^n \mid x^n, t). \tag{92}$$

As with parsing, we assume that $N_{\text{samples}} = 1$. This is also a discrete optimization problem, albeit simpler than parsing, and we again apply branch-and-bound. Similar to the case in parsing, each search state represents a set of derivation trees, represented by an incomplete derivation tree, except that the nodes do not have any sentence positions, since y is not known, and there is only a single logical form rather than a set of logical forms, since x is known. The `branch` function for generation is shown in algorithm 12. The algorithm is started with a derivation tree set whose incomplete derivation tree consists of a single node labeled S with logical form x.

The heuristic upper bound for a search state S is simply:

$$\log h(S) \quad \sum_{m \in S n n} \log p(r^m \mid x^m, t) = \quad , \tag{93}$$

where the sum is over all nodes in the incomplete derivation tree of S, excluding the root node $\mathbf{n}$. Note that, just as in parsing, the algorithm keeps track of this quantity in each search state as the log probability , and so $\log h(S) =$.

Just as in parsing, in order to execute line 15, we can use the augmented branch-and-bound in algorithm 11 to return the m^{th} best derivation tree that maximizes the objective over the set of derivation

Algorithm 12: Pseudocode for `branch` and `expand` in the branch-and-bound algorithm for generating the most likely sentence(s), given a logical form, which aims to maximize equation 92.

```
function branch(derivation tree set S)
    L is an empty list
    n is the root of the incomplete derivation tree of S
    x is the logical form at n
    if n has no child nodes
        A is the nonterminal symbol of n
        return expand(A, x)
    else if n has a nonterminal child node with no children
        A → B_1:f_1 ::: B_K:f_K is the production rule at n
        c_k is the first nonterminal child node of n with no children
        B_k is the nonterminal symbol of c_k
        m is the counter of S
          is the log probability of S
        if f_k(x) fails return ?
        S_m is the m^th most probable derivation tree with root nonterminal B_k and
          logical form f_k(x), according to equation 92
        if S_m exists
            S  = S \ S_m is a new derivation tree set with counter 1, the incomplete
              derivation tree is identical to that of S except c_k is substituted with the
              incomplete derivation tree of S_m, and the log probability is the sum of
              and the log probability of S_m
            L.add(S )
            L.add( a new derivation tree set identical to S except its counter is m + 1)
        else
            A → B_1:f_1 ::: B_K:f_K is the production rule at n
              is the log probability of S
            L.add( a new derivation tree set identical to S except its log probability is
              the sum of  and p(A → B_1:f_1 ::: B_K:f_K | x, t), and is marked COMPLETE )
        return L
function expand(nonterminal A, logical form x)
    L is an empty list
    if A is a preterminal
        for rules A → w do
            S  is a new derivation tree set where the incomplete derivation tree consists
              of a root node n with nonterminal A, logical form x, and child node w
            L.add(S )
    else
        for rules A → B_1:f_1 ::: B_K:f_K do
            S  is a new derivation tree set with counter 1, the incomplete derivation
              tree consists of a root node n with nonterminal A, logical form x, and for
              each child node c_i, the nonterminal is B_i, and logical form is f_i(x)
            if f_i(x) did not fail for all i = 1, ::: , K
                L.add(S )
    return L
```

Sentence: "How large is Alaska?"
Logical form: `answer(A,(size(B,A),const(B,stateid(alaska))))`

Sentence: "How many people lived in Austin?"
Logical form: `answer(A,(population(B,A),const(B,cityid(austin,_))))`

Sentence: "What is the biggest city in Nebraska?"
Logical form:
```
answer(A,largest(A,(city(A),
        loc(A,B),const(B,stateid(nebraska)))))
```

Sentence: "Give me the cities in USA?"
Logical form: `answer(A,(city(A),loc(A,B),const(B,countryid(usa))))`

Figure 14: Examples of sentences and logical form labels from GeoQuery.

trees rooted at B_k with logical form $f_k(x)$. Whenever line 17 is first executed for a given nonterminal B_k and logical form $f_k(x)$, initialize the priority queue Q in algorithm 11 with: `Q.push(S ,h(S ))` where S is the search state with an incomplete derivation tree consisting of a single node at the root with nonterminal B_k and logical form $f_k(x)$. The implementation for our inside-outside sampler, branch-and-bound parser and generator is available at `github.com/asaparov/grammar`.

4.4 SEMANTIC PARSING EXPERIMENTS ON GEOQUERY AND JOBS

To evaluate our parser, we use the GeoQuery and Jobs datasets (Tang and Mooney, 2000; Zelle and Mooney, 1996). GeoQuery contains 880 questions about U.S. geography. Each question is labeled with a logical form in Datalog. The dataset includes a database called GeoBase, which, when each logical form is executed, returns the answer to the corresponding question. The Jobs dataset contains 640 questions about computer-related job postings (from the USENET group `austin.jobs`). Each question is also labeled with a Datalog logical form, similar to the semantic formalism of GeoQuery. Most question in the two datasets are interrogative sentences, but there are some imperative sentences. Figures 14 and 15 showcases some examples from each dataset, respectively. The task is semantic parsing: given each sentence, predict the logical form that represents its meaning.

We created a semantic grammar for the Datalog representation of GeoQuery and Jobs, specifying the "interior" production rules and implementing the semantic transformation functions and their inverses.[2] These experiments are meant to evaluate the language module of PWL, and so the reasoning module is not used as the semantic prior. Rather, we experiment with a simpler prior: Let x be a Datalog logical form, and $x^{a,i}$ is the i^{th} predicate or "function" node in x in prefix order whose smallest variable is a ("smallest" in the sense that A is smaller than B is smaller than C etc). For example, `size(A,B)` is a predicate

2 This grammar is available at `github.com/asaparov/parser/blob/master/english.gram`.

Sentence: "Show me programmer jobs in Tulsa?"
Logical form: `answer(A,(job(A),title(A,T),`
`const(T,'Programmer'),loc(A,C),const(C,'tulsa')))`

Sentence: "What jobs are there with a salary of more than 50000 dollars per year?"
Logical form: `answer(A,(job(A),salary_greater_than(A,50000,year)))`

Sentence: "What jobs in Austin require more than 10 years of experience?"
Logical form: `answer(A,(job(A),loc(A,P),`
`const(P,'austin'),req_exp(A,E),const(E,10)))`

Sentence: "Can I find a job making more than 40000 a year without a degree?"
Logical form: `answer(A,(job(A),`
`salary_greater_than(A,40000,year),\+ req_deg(A)))`

Figure 15: Examples of sentences and logical form labels from JOBS.

node whose smallest variable is `A`, and `most(B,C,...)` is a "function" node whose smallest variable is `B`. The prior probability of x is given by $p(x) \; / \; \prod_{a,i} p(x^{a,i} \mid x^{a,i-1})$ where the conditional $p(x^{a,i} \mid x^{a,i-1})$ is modeled with an HDP as in section 4.1.4. This HDP has height 2: the first feature function is the predicate or "function" symbol of the input node (e.g. `size` or `most`), and the second feature function is the arity and "order" of the arguments (e.g. `size(A)` vs `size(A,B)` vs `size(B,A)`).

We also follow Li, Liu, and Sun (2013), Wong and Mooney (2007), and Zhao and Huang (2015) and experiment with type-checking, where every entity is assigned a type in a type hierarchy (e.g. `alaska` has type `state`, `state` has supertype `polity`, etc), and every predicate is assigned a functional type (e.g. `population` has type `polity ! int ! bool`, etc). We incorporate type-checking into the semantic prior by assigning zero probability to type-incorrect logical forms. More precisely, logical forms are distributed according to the original prior, conditioned on the fact that the logical form is type-correct. Type-checking requires the specification of a type hierarchy. Our hierarchy contains 11 types for GEOQUERY and 12 for JOBS. We run experiments with and without type-checking for comparison.

Following Zettlemoyer and Collins (2007), we use the same 600 GEOQUERY sentences for training and an independent test set of 280 sentences. On JOBS, we use the same 500 sentences for training and 140 for testing. We run our parser with two setups: (1) with no domain-specific supervision, and (2) using a small domain-specific lexicon and a set of beliefs (such as the fact that Portland is a city). For each setup, we run the experiments with and without type-checking, for a total of 4 experimental setups. A given output logical form is considered correct if it is semantically equivalent to the true logical form.[3] In these experiments, we did not use a model of morphology in the grammar. We measure the precision and recall of our method, where

3 The result of execution of the output logical form is identical to that of the true logical form, for any grounding knowledge base/possible world.

precision is the number of correct parses divided by the number of sentences for which our parser provided output, and recall is the number of correct parses divided by the total number of sentences in each dataset . Our results are shown compared against many other semantic parsers in Figure 16. Our method is labeled PWL-LM indicating that while we use the same parser design as the language module of PWM, it uses a distinct grammar and logical formalism (Datalog vs higher-order logic). The numbers for the baselines were copied from their respective papers, and so their specified lexicons/type hierarchies may differ slightly. All code for these experiments is available at `github.com/asaparov/parser`.

Many sentences in the test set contain tokens previously unseen in the training set. In such cases, the maximum possible recall is 88.2 and 82.3 on GeoQuery and Jobs, respectively. Therefore, we also measure the effect of adding a domain-specific lexicon, which maps semantic constants like `maine` to the noun "Maine" for example. This lexicon is analogous to the string-matching and argument identification steps in some other semantic parsers. We constructed the lexicon manually, with an entry for every city, state, river, and mountain in GeoQuery (141 entries), and an entry for every city, company, position, and platform in Jobs (180 entries).

Aside from the lexicon and type hierarchy, the only training information is given by the set of sentences y, corresponding logical forms x, and the domain-independent set of interior production rules, as described in section 4.2.2. In our experiments, we found that the sampler converges rapidly, with only 10 passes over the data. This is largely due to our restriction of the interior production rules to a domain-independent set, which provides significant information about English syntax.

We emphasize that the addition of type-checking and a lexicon are mainly to enable a fair comparison with past approaches. As expected, their addition greatly improves parsing performance. At the time of the publication of our method (Saparov, Saraswat, and Mitchell, 2017), we achieved state-of-the-art F1 on the Jobs dataset. However, even without such domain-specific supervision, the parser performs reasonably well. This is a promising indication that this parser will work effectively in the broader PWL system, and is able to correctly parse sentences with complex and nested semantics. However, we notice a common error is the incorrect determination of scope of functions like `highest`, `shortest`, etc. This is likely due to the fact that the semantic prior does not explicitly model the scope of these functions (it assumes a uniform probability on all possible scopes). Thus, a more explicit model of scope might further improve parsing performance. We found that the semantic parsing problem is easier if the logical forms of each sentence are more similar to the syntactic structure of that sentence. In the extreme case, the logical forms would be identical to the sentences

METHOD	ADDITIONAL SUPERVISION	GEOQUERY			JOBS		
		P	R	F1	P	R	F1
WASP (Wong and Mooney, 2006)	A,B	87.2	74.8	80.5			
-WASP (Wong and Mooney, 2007)	A,B,F	92.0	86.6	89.2			
Extended GHKM (Li, Liu, and Sun, 2013)	A,B,F	93.0	87.6	90.2			
Zettlemoyer and Collins (2005)	C,E,F	**96.3**	79.3	87.0	97.3	79.3	87.4
Zettlemoyer and Collins (2007)	C,E,F	91.6	86.1	88.8			
UBL (Kwiatkowski et al., 2010)	E	94.1	85.0	89.3			
FUBL (Kwiatkowski et al., 2011)	E	88.6	88.6	88.6			
TISP (Zhao and Huang, 2015)	E,F	92.9	**88.9**	**90.9**	85.0	**85.0**	85.0
PWL-LM - lexicon - type-checking	D	86.9	75.7	80.9	89.5	67.1	76.7
PWL-LM + lexicon - type-checking	D,E	88.4	81.8	85.0	91.4	75.7	82.8
PWL-LM - lexicon + type-checking	D,F	89.3	77.9	83.2	93.2	69.3	79.5
PWL-LM + lexicon + type-checking	D,E,F	90.7	83.9	87.2	**97.4**	81.4	**88.7**

Legend for sources of additional supervision:

A. Training set containing 792 examples,
B. Domain-specific set of initial synchronous CFG rules,
C. Domain-independent set of lexical templates,
D. Domain-independent set of interior production rules,
E. Domain-specific initial lexicon,
F. Type-checking and type specification for entities.

Figure 16: Results of semantic parsing experiments on the GEOQUERY and JOBS datasets (Saparov, Saraswat, and Mitchell, 2017). Precision, recall, and F1 scores are shown. The methods in the top portion of the table were evaluated using 10-fold cross validation, whereas those in the bottom portion were evaluated with an independent test set. As a consequence, the methods evaluated using 10-fold cross validation were trained on 792 GEOQUERY examples and tested on 88 examples for each fold (hence the additional supervision label "A" in the above table). In contrast, the methods evaluated using an independent test set were trained on 600 GEOQUERY examples and tested on 280 examples. The domain-independent set of interior production rules (labeled "D" in the above table) is described in section 4.2.2.

Logical form:	`answer(A,smallest(A,state(A)))`	`answer(A,largest(B,(state(A),population(A,B))))`
Test sentence:	"Which state is the smallest?"	"Which state has the most population?"
Generated:	"What state is the smallest?"	"What is the state with the largest population?"

Figure 17: Examples of sentences generated from our trained grammar on logical forms in the GEOQUERY test set (Saparov, Saraswat, and Mitchell, 2017). Generation is performed by computing $\arg\max_{y,t} p(y, t \mid x, t)$ as described in section 4.3.3.

themselves, in which case parsing would be trivial. Thus, there is an inevitable balancing act in designing a semantic grammar and logical formalism for natural language, where on one hand we want the parsing problem to be as simple as possible, but on the other hand, we want the logical forms to be useful for downstream tasks, such as question-answering and reasoning, and ideally the logical forms of two distinct sentences that have the same meaning should be equivalent. These are important lessons to keep in mind when designing a domain-general semantic grammar and logical formalism.

4.5 DOMAIN-GENERAL GRAMMAR AND SEMANTIC FORMALISM

Although we implemented the aforementioned Datalog grammar to be as domain-general as possible, the Datalog representation in GeoQuery and Jobs itself is unable to capture the meaning of many sentences outside the domains of geography and job searching. Consider the sentence "John travels to New York." One possible Datalog logical form for the semantics of this sentence is

```
(const(A,personid(john)),travel(A,B),const(B,cityid('new york',_))).
```

Now consider the sentence "John traveled to New York." While the tense distinction is irrelevant in GeoQuery and Jobs, this is not true in general. Even when time is explicitly specified, as in "John traveled to New York yesterday," it is not clear how to represent this in Datalog. This issue also arises with other adverbial qualifiers, as in "John quickly traveled to New York" or "John will travel to New York after he finishes breakfast." Furthermore, in the above example logical form, the variables `A` and `B` are implicitly universally quantified, according to Datalog semantics. So it is unclear how to represent sentences with existential semantics such as "If I had an airplane, I would fly." These limitations impede Datalog's applicability as a domain-independent representation of natural language semantics.

Other semantic formalisms have similar limitations. For example, while *Abstract Meaning Representation* (AMR) (Banarescu et al., 2013) provides a broad-coverage representation of natural language semantics, it intentionally ignores some central aspects of language in order to simplify and streamline the annotation of a large number of sentences. AMR does not have universal quantification, and so does not distinguish between singular and plural forms of words. The sentences "Alex received a flower" and "Alex received flowers" would have the same logical form, which would be problematic for a downstream task such as answering the question "Did Alex receive at least 2 flowers?" *Discourse Representation Theory* (DRT) (Kamp and Reyle, 1993; Sandt, 1992) is a semantic representation that has explicit universal quantification and is able to capture the semantics of many linguistic phenomena such as anaphora and presupposition. There is a well-defined mapping from DRT logical forms into first-order logic formulas, and so

reasoning over DRT structures is possible using methods developed for first-order logic. Proof calculi for DRT have also been developed to enable reasoning directly with DRT structures (Kamp and Reyle, 1996).

Higher-order logic or *lambda calculus* (Church, 1940) is a well-studied formal language with applications to mathematics and formal semantics. In fact, the field of formal semantics emerged in a large part due to Montague's work on a grammar of English where meaning is represented in higher-order logic (Montague, 1970, 1973, 1974). There are many well-studied proof calculi for performing reasoning in higher-order logic, many of them being extensions of proof calculi for first-order logic, including natural deduction (Gentzen, 1935, 1969). Much of the work in formal semantics built upon Montague's work, including *combinatory categorial grammar* (CCG) (Steedman, 1997).

Thus, we present a new semantic formalism based on higher-order logic and a new grammar with the explicit goal of domain-generality. A *semantic formalism* is a mapping between interpretations of sentences to logical forms. We emphasize that while a language-independent semantic formalism would be tremendously valuable, it is not our primary goal. We focus on representing the semantics of English sentences in this thesis, and we leave extensions to other languages to future work. Higher-order logic provides a convenient way to define *sets* of objects as boolean-valued functions: A boolean-valued function f represents the set of all objects x that make f(x) true. So if f(x) is true if and only if x is a cat, then f is the set of all cats. Higher-order logic provides notation to define new functions, called *function abstraction*, and we can directly use this to "build" new sets:

- x.cat(x) defines the set of all cats,
- x(large(x) ^ dog(x)) is the set of all large dogs,
- x(cat(x) ^ :9y(like(x, y) ^ dog(y))) is the set of all cats that do not like a dog, etc.

Plural noun phrases in English and other languages can express properties of sets of objects, such as "3 books," which refers to a set whose cardinality is three and whose elements are all books. The semantics of this phrase can be represented in higher-order logic as:

$$\texttt{subset}(X,\ \texttt{x.book}(x))\ \texttt{\^{}}\ \texttt{size}(X) = 3,$$

where size is the cardinality function, and subset is defined as:

$$\forall A \forall B(\texttt{subset}(A, B)\ \$\ \forall x(A(x)\ !\ B(x))).$$

That is, subset(A, B) is true if and only if every element of the set A is an element of the set B. Note that the subset is necessary here, since if instead we had written X = x.book(x) ^ size(X) = 3, the set of *all* books would have cardinality three, which is not often the meaning of "3 books." The above tools allows us to analyze almost all noun

phrases as sets. Even singular noun phrases can be analyzed as sets containing only one object: `x.x = alex` is the set that only contains `alex`. This representation enables description of the properties of sets rather than their elements, which, for example, is necessary to represent the meaning of quantity phrases: "at least 3 books," "most books," "a few books," "half of the books on the table," etc.

PARSING VS REASONING: An important lesson from the development of our grammar for Datalog, and from the development of semantic parsers more broadly, is that the semantic parsing problem is easier if the logical forms of each sentence are more similar to the syntactic structure of that sentence. In the extreme case, the logical forms would be identical to the sentences themselves, in which case parsing would be trivial. However, in this extreme case, the resulting logical forms would not be any more useful than the sentences themselves for downstream tasks, such as question-answering and reasoning. At a bare minimum, we want the logical forms represented in a formal language, in which reasoning is amenable (e.g. there is a proof calculus for the formal language). We also want the logical forms of any two sentences with the same meaning to be logically equivalent.

Beyond this bare minimum, there are design options to consider. One such option is the order of commutative operations, such as conjunction. The sentences

- "Alice is a dog and Bob is a cat"
- "Bob is a cat and Alice is a dog"

have the same meaning but differ in their ordering of operands. There are two choices here:

1. The two sentences above have the same logical form: `cat(bob) ^ dog(alice)`. This requires a canonical ordering of operands. In this example, the canonical ordering is determined by the alphabetical ordering of their predicates.
2. The two sentences have distinct but equivalent logical forms, mirroring the order of the respective operands in the sentences: `dog(alice) ^ cat(bob)` for the first sentence, and `cat(bob) ^ dog(alice)` for the second.

In order to implement the first choice in our grammatical framework, the grammar would need two production rules like the following:

```
S ! S:select_left_operand AND:require_canonical
    S:select_right_operand
S ! S:select_right_operand AND:require_canonical
    S:select_left_operand
```

where the function `require_canonical` checks whether the input logical form is a binary conjunction with its operands in canonical order. If so, it simply returns the input logical form unchanged;

otherwise, it returns failure. The function `select_left_operand` returns the left operand of an input conjunction (e.g. given input `dog(alice) ^ cat(bob)`, it returns `dog(alice)`). The function `select_right_operand` returns the right operand of an input conjunction (e.g. given input `dog(alice) ^ cat(bob)`, it returns `cat(bob)`). Similar rules would need to be implemented for any other nonterminal that could be coordinated. This approach scales poorly when the number of operands increases beyond two. In addition, since our parser is top-down, such a grammar would cause a large number of spurious search states to be created. It is for these reasons we chose the second choice: the order of the operands in the logical form matches the order of the corresponding phrases in the sentence. This is not limited to conjunction. For the same reasons as above, we chose that the order of adjuncts of a phrase in the sentence should match the order of the corresponding subexpressions in the logical form. For example, in "She will go there by car in the evening," the subexpression in the logical form corresponding to "by car" should precede the subexpression corresponding to "in the evening." As another example, in "tall green tree," the subexpression in the logical form corresponding to "tall" should precede that which corresponds to "green."

But this design choice does not remove the necessity of canonicalization. It is deferred to the reasoning module. In the above example, it is now the reasoning module's job to determine that `dog(alice) ^ cat(bob)` is equivalent to `cat(bob) ^ dog(alice)`. Thus, these design choices serve to delineate the boundary between the language module and reasoning module. If more computation is deferred to the reasoning module, the language module's job will be easier, at the expense of increasing the complexity of reasoning, and vice versa.

Another design choice in the semantic representation is the representation of named entities. PWM defers named entity linking to the reasoning module (the parser does not do named entity linking). That is, the semantic parser does not parse "Alice" directly into the constant `alice`. Rather, named entities are parsed into existentially-quantified expressions, such as `∃a(name(a) = "Alice" ^ ...)`. This reduces the number of possible logical forms that the parser must consider. If instead the parser were responsible for named entity linking, "Alice" could refer to `alice`, `bob`, or any other concept in the theory. This would dramatically increase the size of the parser's search space.

NEO-DAVIDSONIAN SEMANTICS: Like AMR and DRT, PWL uses *neo-Davidsonian semantics* (Parsons, 1990) (also known as *event semantics*) to represent meaning of actions and events in all logical forms (both in the theory and during semantic parsing). As a concrete example, a straightforward way to represent the meaning of "Jason traveled to New York" could be with the logical form `travel(jason, nyc)`. In neo-Davidsonian semantics, this would instead be represented with

	Semantic parser output	Possible axioms in theory
Without neo-Davidsonian semantics, language module performs entity linking	$\exists b(\texttt{book}(b) \wedge \texttt{write}(\texttt{alex}, b))$	$\texttt{book}(c_1)$, $\texttt{write}(\texttt{alex}, c_1)$.
With neo-Davidsonian semantics, language module performs entity linking	$\exists b(\texttt{book}(b) \wedge \exists w(\texttt{arg1}(w) = \texttt{alex} \wedge \texttt{write}(w) \wedge \texttt{arg2}(w) = b))$	$\texttt{book}(c_1)$, $\texttt{write}(c_2)$, $\texttt{arg1}(c_2) = \texttt{alex}$, $\texttt{arg2}(c_2) = c_1$.
With neo-Davidsonian semantics, reasoning module resolves named entities (our approach)	$\exists a(\texttt{name}(a) = \text{“Alex”} \wedge \exists b(\texttt{book}(b) \wedge \exists w(\texttt{arg1}(w) = a \wedge \texttt{write}(w) \wedge \texttt{arg2}(w) = b)))$	$\texttt{book}(c_1)$, $\texttt{write}(c_2)$, $\texttt{name}(c_3) = \text{“Alex”}$, $\texttt{arg1}(c_2) = c_3$, $\texttt{arg2}(c_2) = c_1$.

Table 2: Design choices in the representation of the meaning of the sentence “Alex wrote a book.” To avoid clutter, atoms that convey tense/aspect information are omitted from the logical forms.

three distinct atoms: $\texttt{travel}(c_1)$, $\texttt{arg1}(c_1) = \texttt{jason}$, and $\texttt{arg2}(c_1) = \texttt{nyc}$. Here, c_1 is a constant that represents the “traveling event,” whose first argument is the constant representing Jason, and whose second argument is the constant representing New York City. This representation allows the event to be more readily modified by other logical expressions, such as in “Jason quickly traveled to NYC with my car before nightfall.” Neo-Davidsonian semantics is not a full semantic formalism in its own right, in that it does not prescribe a complete logical form for the meaning of every sentence. Rather, it is a way to represent events and actions (often expressed as verbs in natural language) in a semantic formalism, where predicates are reified as objects with properties. Table 2 illustrates the design choices in the representation of named entities and neo-Davidsonian semantics. Note that in the subexpression of the logical form representing “Alex wrote a book”:

$$\exists w(\texttt{arg1}(w) = a \wedge \texttt{write}(w) \wedge \texttt{past}(w) \wedge \texttt{arg2}(w) = b),$$

the order of the conjuncts mirrors the order of the corresponding words in the sentence: a is an entity named “Alex” and b is an instance of a $\texttt{book}$.

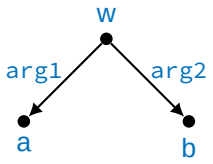
Every inflected verb has a corresponding atom in the logical form that indicates its tense and aspect. This atom always follows the atom that declares the type of the event. In the above example, “wrote” is in the simple past tense, which is represented as $\texttt{past}(w)$ and immediately follows $\texttt{write}(w)$. There are 12 predicates to convey tense-aspect: one for every combination of tense (past, present, future) and aspect (simple, perfect, progressive, and perfect progressive). For example

`future_perfect_progressive` represents future tense and perfect progressive aspect (e.g. "will have been writing").

SEMANTIC VS SYNTACTIC HEAD: The *syntactic head* of a phrase is the word that determines the syntactic category of that phrase. For example, the head of the noun phrase "the apple that fell from the tree" is "apple," and the head of the adjectival phrase "exceedingly bright" is "bright." For any logical form representation of a natural language phrase, we define the *semantic head* as the subexpression within the logical form that corresponds to the syntactic head of the phrase. Consider the example "Alex wrote a book." It's logical form is

```
∃a(name(a) = "Alex" ∧ ∃b(book(b)
    ∧ ∃w(arg1(w) = a ∧ write(w) ∧ past(w) ∧ arg2(w) = b))).
```

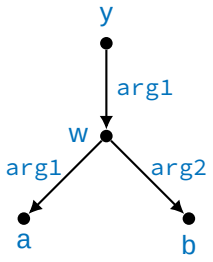
The syntactic head is "wrote" and the semantic head is the subexpression containing the existential quantification over `w`. In our new semantic formalism, *all* logical forms have a semantic head. Observe that the argument relations between the variables in the logical form define a directed graph over the variables: Every variable corresponds to a vertex, and every atom `arg1(x) = y` and `arg2(x) = y` corresponds to a labeled edge from `x` to `y`. In the above example, there are three variables: `a`, `b`, and `w`; and two edges: an edge `(w, a)` labeled `arg1`, and an edge `(w, b)` labeled `arg2`. This *scope graph* is shown below:



The semantic head is identified as the *source* of this graph (i.e. the vertex with no incoming edges). But this relationship does not extend to the example "Alex wrote a book yesterday," which has the logical form

```
∃a(name(a) = "Alex" ∧ ∃b(book(b)
    ∧ ∃w(arg1(w) = a ∧ write(w) ∧ past(w) ∧ arg2(w) = b
    ∧ ∃y(yesterday(y) ∧ arg1(y) = w)))).
```

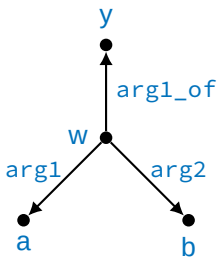
But the scope graph for this logical form is



This would imply that the semantic head is y, corresponding to "yesterday." To avoid this, like AMR, we define inverses for the argument functions: arg1_of and arg2_of, where arg1_of(a) = b if and only if arg1(b) = a. With these functions, we can rewrite the logical form for "Alex wrote a book yesterday" as

$$\exists a(\texttt{name}(a) = \text{"Alex"} \wedge \exists b(\texttt{book}(b) \wedge \exists w(\texttt{arg1}(w) = a \wedge \texttt{write}(w) \wedge \texttt{past}(w) \wedge \texttt{arg2}(w) = b \wedge \exists y(\texttt{yesterday}(y) \wedge \texttt{arg1\_of}(w) = y)))),$$

and the resulting scope graph is



The source vertex of the graph is w, which correctly corresponds to the syntactic head of the sentence "wrote."

In general, the following algorithm is used to find the semantic head, starting with the root of the logical form:

1. Check if the current subexpression is the head:
   - If the nonterminal is not a noun phrase, and the current subexpression is an existentially-quantified conjunction $\exists x(\ldots \wedge t(x) \wedge \ldots)$ that contains an atom which declares the type of x, t(x), and t is not a "special" predicate, and there is no term of the form arg1(⋆) = x, arg2(⋆) = x, arg1_of(⋆) = x, or arg2_of(⋆) = x, then return this node as the head. "Special" predicates are arg1, arg2, arg1_of, arg2_of, size, and any aspect-tense predicate.
   - If the nonterminal is a noun phrase, and the current expression is a declaration of a set: $\exists X(F(X) \wedge \ldots \wedge \exists x(X(x) \wedge Q(x)))$ or $\exists X(F(X) \wedge \ldots \wedge \forall x(X(x) \; ! \; Q(x)))$ where F(X) is a definition of the set X, then return this node as the head. Examples of set definitions are X = x.cat(x) and subset(X, x.cat(x)).
2. If the current subexpression is not the head, then continue this procedure recursively on the right-most child subexpression (i.e. if this is a conjunction, repeat with the right-most operand; if this is an if-then expression, repeat with the consequent; etc).

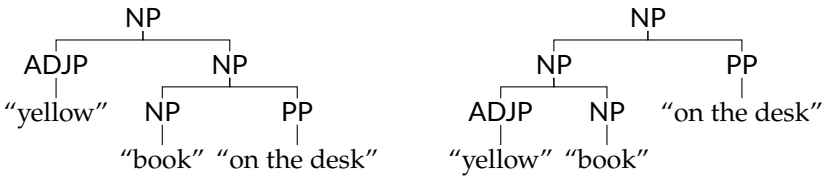
Observe that applying the above procedure to the example logical forms for "Alex wrote a book" and "Alex wrote a book yesterday" will correctly return the subexpression with the existential quantification over w. The notion of the semantic head is useful since the semantic transformation functions operate on the semantic head of the logical form.

After parsing a sentence into logical form, and before passing it onto the reasoning module, PWL will convert these inverse argument functions into their regular form: `arg1_of(a) = b` is converted into `arg1(b) = a`, and `arg2_of(a) = b` is converted into `arg2(b) = a`.

Another very important lesson learned from the experiments on GeoQuery and Jobs is that the following is a very desirable property of a grammar in this framework: For any natural language phrase y and any nonterminal A *2* N, there is a one-to-one correspondence between logical form meanings and derivation trees of y with root N. That is, for any logical form interpretation x of a natural language phrase y and for any nonterminal A *2* N, there is no more than one derivation tree with root N of that phrase y with that logical form x. Under this property, there is no ambiguity in derivations given the logical form and nonterminal. A consequence of this property is that during parsing, once the algorithm has found a logical form parse for a given phrase and nonterminal, we would guaranteed that there are no other derivations of that phrase that has the same logical form. This has the effect of greatly reducing the number of search states that the parser has to visit, increasing the performance of the parser overall. One common source of ambiguity that violates this property arises from production rules that combine adjuncts. Consider the following example production rules (omitting semantic transformation functions):

NP *!* NP PP, NP *!* ADJP NP.

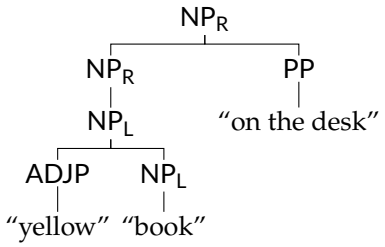
And consider the phrase "yellow book on the desk." With the above grammar, the unique derivation property would be violated since there are two ambiguous derivations for the same logical form:



To avoid this ambiguity, the nonterminal NP can instead be split into two: NP_L and NP_R. The nonterminal NP_L is given the production rules for the left adjuncts, whereas the nonterminal NP_R is given the production rules for the right adjuncts:

NP_R *!* NP_R PP, NP_R *!* NP_L, NP_L *!* ADJP NP_L.

With these modified production rules, "yellow book on the desk" is no longer ambiguous:



In the grammar implemented by PWL, we similarly split the nonterminals for verb phrases, noun phrases, adjectival phrases, and adverbial phrases into "left" and "right" counterparts. We also added code in our parser to detect when a duplicate logical form is found for any nonterminal and span of the sentence. This facilitates debugging the inverse semantic transformation functions and maintains the performance of the parser by avoiding unintentionally increasing the size of the parser's search space.

We designed the production rules of the grammar following many of the ideas of Huddleston and Pullum (2002). Our semantic grammar framework enabled us to look to the rich literature in the field of formal semantics and incorporate ideas thereof into the grammar, such as neo-Davidsonian semantics. As another example, consider the phrase "two cups of water." Rothstein (2010) posits that there are two interpretations: The first interpretation is a quantity of water whose volume is equal to roughly 473.18 milliliters. The second interpretation is two containers with handles, each filled with water. Note that in the first interpretation, the water may all be inside a single container, whereas in the second interpretation, the total volume of the water may be very different from 473.18 milliliters if the cups are especially small or large. The first interpretation is known as the "measure" reading, and the second is known as the "count" reading. In our logical formalism, the two readings are shown for the sentence "I drank two cups":

$$\exists c(\texttt{cup\_unit}(c) \wedge \exists m(\texttt{measure}(m) \wedge \texttt{arg1}(m) = 2 \wedge \texttt{arg2}(m) = c \wedge \exists d(\texttt{arg1}(d) = \texttt{me} \wedge \texttt{drink}(d) \wedge \texttt{past}(d) \wedge \texttt{arg2}(d) = m))),$$

$$\exists C(C = c.\texttt{cup}(c) \wedge \texttt{size}(C) = 2 \wedge \forall c(C(c) \rightarrow \exists d(\texttt{arg1}(d) = \texttt{me} \wedge \texttt{drink}(d) \wedge \texttt{past}(d) \wedge \texttt{arg2}(d) = c))).$$

In our grammar, the count reading is always available for noun phrases that contain a non-negative integer followed by a nominal. To allow for the other reading, we added a production rule to our grammar NP $\rightarrow$ Q NOMINAL$_R$ (semantic transformation functions not shown) that requires the input logical form have a `measure` event at the head, the first argument of this event is passed to the first child (Q) and the second argument is passed to the second child (NOMINAL$_R$).

The predicate `same` represents the meaning of the verb "to be," indicating the equivalence of two objects, such as in "Pennsylvania is a state." After parsing sentences into logical forms, and before giving

them to the reasoning module, PWL simplifies expressions of the form $\exists x(\texttt{same}(x) \wedge \texttt{arg1}(x) = a \wedge \texttt{arg2}(x) = b)$ into $a = b$. The order of the conjuncts inside the $\exists x$ does not matter.

The passive voice is represented using a higher order predicate `inverse`, such as in the logical form for "A book was written by Alex:"

$$\exists b(\texttt{book}(b) \wedge \exists a(\texttt{name}(a) = \text{“Alex”} \wedge \exists w(\texttt{arg1}(w) = b \wedge \texttt{inverse}(\texttt{write})(w) \wedge \texttt{past}(w) \wedge \texttt{arg2}(w) = a))).$$

This allows us to maintain `arg1` as the indicator of the subject of the verb, and `arg2` as the indicator of the object of the verb, thereby alleviating the need for additional production rules in the grammar. After parsing, PWL simplifies expressions of the form $\exists x(\texttt{inverse}(f)(x) \wedge \texttt{arg1}(x) = a \wedge \texttt{arg2}(x) = b)$ into $\exists x(f(x) \wedge \texttt{arg2}(x) = a \wedge \texttt{arg1}(x) = b)$, swapping the arguments. The order of the conjuncts or the presence of additional conjuncts inside the $\exists x$ does not matter. So in the above logical form for "A book was written by Alex," the logical form would be simplified into:

$$\exists b(\texttt{book}(b) \wedge \exists a(\texttt{name}(a) = \text{“Alex”} \wedge \exists w(\texttt{arg2}(w) = b \wedge \texttt{write}(w) \wedge \texttt{past}(w) \wedge \texttt{arg1}(w) = a))).$$

Our new grammar is available at `github.com/asaparov/PWL/blob/main/english.gram`.

4.5.1 *Intra-sentential coreference*

Anaphora is widely prevalent in natural language, and wide-coverage parsing requires a way to resolve anaphora. However, since PWM makes the assumption that each sentence is context-independent (the sentences are conditionally independent given the theory), it is not possible to correctly model cross-sentential anaphora. In order to do so, we would need a model of context, which we propose in section 4.7 along with other future work. But for the sake of rapid prototyping, we instead use a simpler model of anaphora, which we present as follows.

As with named entity linking, the language module of PWL does not perform coreference/anaphora resolution. Rather, anaphora are semantically interpreted as objects with special types: `ref` ("it"), `female_ref` ("she" and "her"), `male_ref` ("he" and "him"), and `plural_ref` ("they" and "them"). For example, the logical form of "He saw her" is represented as

$$\exists a(\texttt{male\_ref}(a) \wedge \exists b(\texttt{female\_ref}(b) \wedge \exists s(\texttt{arg1}(s) = a \wedge \texttt{see}(s) \wedge \texttt{past}(s) \wedge \texttt{arg2}(s) = b))).$$

If this logical form were directly given to the reasoning module, the variables a and b could be instantiated as any constant in the theory,

regardless of how recently that constant is mentioned in the sequence of sentences. In natural language, anaphora are much more likely to bind to more recently mentioned entities. Therefore, in order to incorporate this strong recency preference in anaphora binding, we alter the generative process of PWM slightly: Each logical form x_i is generated in the same way as before. These logical forms do not have any special anaphoric types (i.e. `ref`, `female_ref`, `male_ref`, etc). Rather than proceeding to generate sentences directly from these logical forms x_i, we introduce an intermediate step, where for each x_i, a new logical form $x_i^{\emptyset}$ is generated where some variables are replaced by new existentially quantified variables with anaphoric types. The sentences y_i are then generated from these new logical forms $x_i^{\emptyset}$. As an example, consider the non-anaphoric logical form x_i

$$\begin{aligned}&\forall p(\texttt{planet}(p) \wedge \exists s(\texttt{star}(s)\\ &\quad \wedge \exists n(\texttt{arg1}(n) = p \wedge \texttt{near}(n) \wedge \texttt{arg2}(n) = s))\\ &\quad \rightarrow \exists h(\texttt{arg1}(h) = p \wedge \texttt{hot}(h))).\end{aligned} \tag{94}$$

The intermediate (anaphoric) logical form $x_i^{\emptyset}$ could remain unchanged, or it could introduce anaphora. If unchanged, the resulting sentence could be something like "Every planet near a star is hot," or "Every planet that is near a star is hot." If anaphora is introduced, an example intermediate logical form $x_i^{\emptyset}$ is

$$\begin{aligned}&\exists p(\texttt{planet}(p) \wedge \exists s(\texttt{star}(s)\\ &\quad \wedge \exists n(\texttt{arg1}(n) = p \wedge \texttt{near}(n) \wedge \texttt{arg2}(n) = s)))\\ &\quad \rightarrow \exists r(\texttt{ref}(r) \wedge \exists h(\texttt{arg1}(h) = r \wedge \texttt{hot}(h))),\end{aligned} \tag{95}$$

which could result in a sentence such as "If a planet is near a star, it is hot." Notice the quantifier $\forall p$ was changed into $\exists p$ as it was moved from the root scope into the scope of the antecedent (the left side of $\rightarrow$). This is due to the fact that under classical logic, $\forall x(f(x) \rightarrow g)$ is equivalent to $(\exists x.f(x)) \rightarrow g$. But the quantifier is unchanged if moved into the consequent. Similarly, the quantifier is changed if it is moved into a negation: $\exists x. : f(x)$ is equivalent to $: \forall x.f(x)$. The conditional probability of the anaphoric logical form, given the non-anaphoric logical form, is

$$p(x_i^{\emptyset} \mid x_i) \propto \exp\left\{-\sum_{a \in \text{anaphora}(x_i^{\emptyset})} \text{dist}_{x_i}(a)\right\}, \tag{96}$$

where the sum is taken over variables with anaphoric types (in the example logical form in equation 95, there is only one such variable: r), and $\text{dist}_{x_i}(a)$ is the "distance" between the anaphora and the referent scope. This distance is defined as follows: create a sorted list of non-event variables according to the prefix ordering of variables in the logical form $x_i^{\emptyset}$. The distance between an anaphoric variable

and its referent variable is simply the distance of the variables in the sorted list. For example, in the logical form in equation 95, the sorted list of non-event variables is: p, s, and r. The distance between the anaphoric variable r and its referent p is $\text{dist}_{x_i}(r) = 2$. Since our logical formalism was designed so that the order of the elements in the logical form closely mirrors the order of the corresponding tokens in the sentence, this notion of distance between elements in the logical form corresponds closely to the notion of distance between the corresponding tokens in the sentence. While this simple conditional distribution incorporates the fact about anaphora binding that more recently mentioned referents are preferred, it ignores other known principles of *binding theory* in linguistics, such as the fact that subject entities are more likely to be referents than object entities ("planet" vs "star" in the above example). We suggest a different model as future work in section 4.7 that could better incorporate such principles into the model.

During inference, PWL effectively performs the inverse of the generative process: first parse the sentence y_i into the top-k values of the anaphoric logical form x_i^0, then for each of the top-k logical forms, resolve the anaphora to produce the top-k^0 values of the non-anaphoric logical form x_i. These non-anaphoric logical forms x_i are then given to the reasoning module.

4.5.2 *Data structure for sets of logical forms*

The algorithms in section 4.3 operate over *sets* of logical forms. Some of these sets can be infinite. The starting search state for these algorithms is often the set of all logical forms. Thus, it is not feasible to represent each set as a list of logical forms. A memory-efficient data structure to represent sets of logical form is needed, which we present in this section.

This data structure needs to implement the *set intersection* operation: given two sets of logical forms X and Y, compute $X \setminus Y$. In addition, the semantic transformation functions, their inverses, and semantic feature functions work with logical form sets, and so the data structure must also provide the operations needed by the implementations thereof. Many of the semantic transformation functions in our new grammar, as well as `exclude_features` as discussed in section 4.1.4, require the *set difference* operation: given two sets of logical form X and Y, compute $X \cap Y$. Recall that the parser relies on a subroutine to compute $f^{-1}(X) \setminus Y$ where f is a semantic transformation function, $f^{-1}(X) = \{x : f(x) \in X\}$ is its inverse, and X and Y are sets of logical forms (see line 20 in algorithm 9). This subroutine, as well as the set intersection operation, is permitted to return a list of sets that represents their union: $f^{-1}(X) \setminus Y = Z_1 \cup \cdots \cup Z_r$. These sets $Z_1, \ldots, Z_r$ should be disjoint, since otherwise it is possible for the parser to visit search

states more than once and can lead to wasted computation. We will show that the set difference operation helps to ensure that the output of set intersection is disjoint: $Z_1, (Z_2 \cap Z_1), \ldots, (Z_r \cap Z_1 \cap \ldots \cap Z_{r-1})$.

The core of our data structure is actually identical to that for a singleton logical form in higher-order logic, and is shown below in algorithm 13 in a pseudo-programming language with subtyping polymorphism.

Algorithm 13: Pseudocode for standard data structure representing a higher-order logic formula.

```
1  class hol_term
       /* supertype that represents the
          higher-order formula */
2  class hol_not extends hol_term
3      hol_term operand
4  class hol_if_then extends hol_term
5      hol_term antecedent
6      hol_term consequent
7  class hol_equals extends hol_term
8      hol_term left
9      hol_term right
10 class hol_and extends hol_term
11     array<hol_term> operands
12 class hol_or extends hol_term
13     array<hol_term> operands
14 class hol_for_all extends hol_term
15     int variable
16     hol_term operand
17 class hol_exists extends hol_term
18     int variable
19     hol_term operand
20 class hol_lambda extends hol_term
21     int variable
22     hol_term operand
23 class hol_application extends hol_term
       /* a function application (e.g. arg1(X)) */
24     hol_term function
25     array<hol_term> arguments
26 class hol_true extends hol_term
       /* subtype representing > */
27 class hol_false extends hol_term
       /* subtype representing ? */
28 class hol_variable extends hol_term
29     int variable
30 class hol_constant extends hol_term
31     int constant
32 class hol_string extends hol_term
33     string str
34 class hol_number extends hol_term
35     number num
```

Each logical form is an instance of the type `hol_term`. To extend this to represent sets of logical forms, we add a new "wildcard" subtype `hol_any_right`:

```
36 class hol_any_right extends hol_term
37     hol_term? included    /* can be null */
38     array<hol_term> excluded
```

This structure represents the set of all higher-order expressions, which are not elements of any `excluded` set, and have a subexpression in the right-most path which is an element of the logical form set corresponding to the field `included` (the right-most path of an expression tree is the set of nodes visited by starting from the root and walking to the right-most child at each node). To be precise, if x is a logical form, $r(x)$ is the right-most child node of x (i.e. if $x = x_1 \wedge \ldots \wedge x_n$ is a conjunction, $r(x) = x_n$ is the right-most operand; if $x = x_1 \Rightarrow x_2$ is an if-then expression, $r(x) = x_2$ is the consequent subexpression; if $x = \forall y.A$ is a quantified expression, $r(x) = A$ is the quantificand; etc).

Define S_R as the right-most path in the expression tree of x. That is, the smallest set such that:

$$x \in S_R(x), \text{ the root node is an element of } S_R(x), \tag{97}$$

$$\text{and for any } n \in S_R(x), r(n) \in S_R(x). \tag{98}$$

Then for any logical form set X, define $R(X)$ as the set of all logical forms that have a subexpression in the right-most path that is an element of X:

$$R(X) \quad , \quad \{x : \exists n \in S_R(x) \text{ such that } n \in X\}. \tag{99}$$

With this notation, we can precisely define the semantics of the `hol_any_right` structure: let X be the logical form set that corresponds to the `included` field, and $Y_1, \ldots, Y_n$ be the logical form sets corresponding to the `excluded` field. Then, the `hol_any_right` structure represents the set of logical forms

$$R(X) \cap (Y_1 \cup \ldots \cup Y_n). \tag{100}$$

Note that `included` can be *null*, in which case the structure represents the set of logical forms

$$\cap (Y_1 \cup \ldots \cup Y_n), \tag{101}$$

where $\cap$ is the set of *all* logical forms. So if both `included` is *null* and `excluded` is empty, then the data structure represents the unconstrained set of all logical forms $\cap$. We require that `included` is not a subset of the `excluded` sets (i.e. $R(X) * Y_1 \cup \ldots \cup Y_n$), since otherwise the resulting set would be empty. We also require that no `excluded` set is superfluous (i.e. $Y_i \setminus R(X)$ is not empty for all i).

Since `hol_any_right` is a subtype of `hol_term`, it can appear as a subexpression within larger expressions. For example, the expression

$$\exists b(\texttt{book}(b) \wedge R(\exists w(\texttt{write}(w) \wedge \texttt{past}(w) \wedge \texttt{arg2}(w) = b)))$$

represents the set of all logical forms that look like $\exists b(\texttt{book}(b) \wedge \ldots)$ where the $\ldots$ is any logical form that contains the subexpression $\exists w(\texttt{write}(w) \wedge \texttt{past}(w) \wedge \texttt{arg2}(w) = b))$ in its right-most path. As another example, $R(a) \wedge R(3)$ is the set of all logical forms with a binary conjunction at the root, where the left child is any logical form with the constant a as its right-most descendant node, and the right child is any logical form with the number 3 as its right-most descendant node.

The `hol_any_right` structure was chosen to reflect the fact that the semantic head of a logical form in our new formalism is a subexpression in the right-most path of the expression tree. Recall that the semantic transformation functions in our grammar operate on the semantic head of the input logical form. So we can use `hol_any_right`

Algorithm 14: Pseudocode to compute the set intersection of two logical form sets, each represented by the `hol_term` data structure.

```
function set_intersect(hol_term X, hol_term Y)
    if X has type hol_any_right
        return set_intersect_any_right(X, Y)
    else if Y has type hol_any_right
        return set_intersect_any_right(Y, X)
    else if X and Y have different types  return empty list
    L is an empty list
    if X has type hol_not
        [Z_1, ..., Z_n] = set_intersect(X.operand, Y.operand)
        for i = 1, ..., n do  L.add( : Z_i)
    else if X has type hol_if_then
        [Z^L_1, ..., Z^L_n] = set_intersect(X.antecedent, Y.antecedent)
        [Z^R_1, ..., Z^R_m] = set_intersect(X.consequent, Y.consequent)
        for i = 1, ..., n do
            for j = 1, ..., m do  L.add(Z^L_i ! Z^R_j)
    else if X has type hol_equals
        [Z^L_1, ..., Z^L_n] = set_intersect(X.left, Y.left)
        [Z^R_1, ..., Z^R_m] = set_intersect(X.right, Y.right)
        for i = 1, ..., n do
            for j = 1, ..., m do  L.add(Z^L_i = Z^R_j)
    else if X has type hol_and or hol_or
        if X.operands.length ≠ Y.operands.length  return empty list
        let N be X.operands.length
        for i = 1, ..., N do
            [Z^i_1, ..., Z^i_{n_i}] = set_intersect(X.operands[i], Y.operands[i])
        for f(i_1, ..., i_N) : i_j ∈ f1, ..., n_j} do
            if X has type hol_and  L.add(Z^1_{i_1} ^ ... ^ Z^N_{i_N})
            if X has type hol_or  L.add(Z^1_{i_1} _ ... _ Z^N_{i_N})
    else if X has type hol_for_all or hol_exists or hol_lambda
        if X.operands.variable ≠ Y.operands.variable  return empty list
        [Z_1, ..., Z_n] = set_intersect(X.operand, Y.operand)
        for i = 1, ..., n do
            let x be X.operands.variable
            if X has type hol_for_all  L.add(∀x.Z_i)
            if X has type hol_exists  L.add(∃x.Z_i)
            if X has type hol_lambda  L.add( x.Z_i)
    else if X has type hol_func_application
        if X.arguments.length ≠ Y.arguments.length  return empty list
        [Z^0_1, ..., Z^0_n] = set_intersect(X.function, Y.function)
        let N be X.arguments.length
        for i = 1, ..., N do
            [Z^i_1, ..., Z^i_{n_i}] = set_intersect(X.arguments[i], Y.arguments[i])
        for f(i_1, ..., i_N) : i_j ∈ f1, ..., n_j} do
            for i = 1, ..., n do  L.add(Z^0_i(Z^1_{i_1}, ..., Z^N_{i_N}))
    else if X has type hol_true  L.add(>)
    else if X has type hol_false  L.add(?)
    else if X has type hol_variable and X.variable = Y.variable
        L.add(X)
    else if X has type hol_constant and X.constant = Y.constant
        L.add(X)
    else if X has type hol_string and X.str = Y.str
        L.add(X)
    else if X has type hol_number and X.num = Y.num
        L.add(X)
    return L
```

to specify the semantic head of logical forms in a set, even when there are no constraints on other parts of the logical form in that set. This is related to the more general fact that the data structure design for logical form sets is intimately linked to the design of the semantic transformation functions of the grammar, and therefore, the design of the logical formalism.

SET INTERSECTION: With the above data structure, our algorithm for computing the set intersection of any two sets of logical forms is shown in algorithm 14. In this algorithm, the loop over $f(i_1, :::, i_N) : i_j \ 2 \ f1, :::, n_j g$ on lines 26 and 43 is a loop over all tuples $(i_1, :::, i_N)$ such that for each j, i_j is an integer between 1 and n_j inclusive (i.e. the Cartesian product $f1, :::, n_1 g \quad f1, :::, n_2 g \quad ::: \quad f1, :::, n_N g$). If either of the two input logical form sets have type `hol_any_right`, it calls the helper function `set_intersect_any_right`, which is shown in algorithm 15.

The correctness of `set_intersect_any_right` in the case where Y has type `hol_any_right` (on lines 4-18 in algorithm 15) relies on the following fact:

Thm 2. *Given any sets of logical forms* A^X, B^X, A^Y, *and* B^Y,

$$(R(A^X) \cap B^X) \setminus (R(A^Y) \cap B^Y) = (R(A^X \setminus R(A^Y)) \cap (B^X \cup B^Y)) \cup (R(A^Y \setminus R(A^X)) \cap (B^X \cup B^Y)).$$

Proof. Note that

$$(R(A^X) \cap B^X) \setminus (R(A^Y) \cap B^Y) = R(A^X) \setminus R(A^Y) \setminus \overline{B^X} \setminus \overline{B^Y},$$
$$= (R(A^X) \setminus R(A^Y)) \cap (B^X \cup B^Y).$$

So it will suffice to show that $R(A^X) \setminus R(A^Y) = R(A^X \setminus R(A^Y)) \cup R(A^Y \setminus R(A^X))$.

Take any $x \ 2 \ R(A^X) \setminus R(A^Y)$. By definition, there is a subtree $a \ 2 \ S_R(x)$ such that $a \ 2 \ A^X$ and there is a subtree $b \ 2 \ S_R(x)$ such that $b \ 2 \ A^Y$. There are two cases: $a \ 2 \ S_R(b)$ or $b \ 2 \ S_R(a)$.

1. If $a \ 2 \ S_R(b)$, then $b \ 2 \ R(A^X)$, and therefore, $b \ 2 \ A^Y \setminus R(A^X)$ which implies $x \ 2 \ R(A^Y \setminus R(A^X))$.

2. If instead $b \ 2 \ S_R(a)$, then $a \ 2 \ R(A^Y)$, and therefore, $a \ 2 \ A^X \setminus R(A^Y)$ which implies $x \ 2 \ R(A^X \setminus R(A^Y))$.

We conclude that $R(A^X) \setminus R(A^Y) \quad R(A^X \setminus R(A^Y)) \cup R(A^Y \setminus R(A^X))$.

To show the other direction, take any $x \ 2 \ R(A^X \setminus R(A^Y)) \cup R(A^Y \setminus R(A^X))$.

1. In the first case, $x \ 2 \ R(A^X \setminus R(A^Y))$, which means there is a subtree $a \ 2 \ S_R(x)$ such that $a \ 2 \ A^X \setminus R(A^Y)$. This implies that $x \ 2 \ R(A^X)$, and that there is a subtree $b \ 2 \ S_R(a)$ such that $b \ 2 \ A^Y$. By definition, b is also a member of $S_R(x)$, and so $x \ 2 \ R(A^Y)$.

Algorithm 15: Helper function for computing the intersection of two sets of logical forms, where X has type `hol_any_right`.

```
function set_intersect_any_right(hol_any_right X, hol_term Y)
    L is an empty list
    let R(A^X) n (B_1^X [ ::: [ B_n^X) be the set represented by X
    if Y has type hol_any_right
        let R(A^Y) n (B_1^Y [ ::: [ B_m^Y) be the set represented by Y
        if A^X = A^Y
            if R(A^X)  B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y  return empty list
            return [R(A^X) n (B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y)]
        [Z_1, ::: , Z_s] = set_intersect_any_right(R(A^X), A^Y)
        [W_1, ::: , W_t] = set_intersect_any_right(R(A^Y), A^X)
        for i = 1, ::: , s do
            if R(Z_i)  B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y  continue
            L.add(R(Z_i) n (B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y))
        for i = 1, ::: , t do
            if R(W_i)  B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y [ R(Z_1) [ ::: [ R(Z_s)
                continue
            L.add(R(W_i) n (B_1^X [ ::: [ B_n^X [ B_1^Y [ ::: [ B_m^Y [ R(Z_1) [ ::: [ R(Z_s)))
        return L
    let K be a list initially containing only Y
    for i = 1, ::: , n do
        K  is an empty list
        for each Y^0 in K do
            K .add_all(set_subtract(Y^0, B_i^X))
        set K = K
    for each Y^0 in K do
        if Y^0 has type hol_not
            [Z_1, ::: , Z_s] = set_intersect_any_right(R(A^X), Y^0.operand)
            for i = 1, ::: , s do L.add(:Z_i)
            [W_1, ::: , W_s] = set_intersect(A^X, Y^0)
            for i = 1, ::: , s do
                [W_1^0, ::: , W_t^0] = set_subtract(W_i.operand, R(A^X))
                for j = 1, ::: , t do L.add(:W_j^0)
        else if Y^0 has type hol_if_then
            [Z_1, ::: , Z_s] = set_intersect_any_right(R(A^X), Y^0.consequent)
            for i = 1, ::: , s do L.add(Y^0.antecedent ! Z_i)
            [W_1, ::: , W_s] = set_intersect(A^X, Y^0)
            for i = 1, ::: , s do
                [W_1^0, ::: , W_t^0] = set_subtract(W_i.consequent, R(A^X))
                for j = 1, ::: , t do L.add(Y^0.antecedent ! W_j^0)
        else if Y^0 has type hol_equals
            [Z_1, ::: , Z_s] = set_intersect_any_right(R(A^X), Y^0.right)
            for i = 1, ::: , s do L.add(Y^0.left = Z_i)
            [W_1, ::: , W_s] = set_intersect(A^X, Y^0)
            for i = 1, ::: , s do
                [W_1^0, ::: , W_t^0] = set_subtract(W_i.right, R(A^X))
                for j = 1, ::: , t do L.add(Y^0.left = W_j^0)
```

Algorithm 15: (continued)

```
    else if Y^0 has type hol_and or hol_or
        let [Y^0_1, ..., Y^0_N] be the operands of Y^0
        [Z_1, ..., Z_s] = set_intersect_any_right(R(A^X), Y^0_N)
        for i = 1, ..., s do
            if Y^0 has type hol_and  L.add(Y^0_1 ^ ... ^ Y^0_{N-1} ^ Z_i)
            if Y^0 has type hol_or   L.add(Y^0_1 _ ... _ Y^0_{N-1} _ Z_i)
        [W_1, ..., W_s] = set_intersect(A^X, Y^0)
        for i = 1, ..., s do
            let [S_1, ..., S_N] be the operands of W_i
            [W^0_1, ..., W^0_t] = set_subtract(S_N, R(A^X))
            for j = 1, ..., t do
                if Y^0 has type hol_and  L.add(Y^0_1 ^ ... ^ Y^0_{N-1} ^ W^0_j)
                if Y^0 has type hol_or   L.add(Y^0_1 _ ... _ Y^0_{N-1} _ W^0_j)
    else if Y^0 has type hol_for_all or hol_exists or hol_lambda
        let x be Y^0.variable
        [Z_1, ..., Z_s] = set_intersect_any_right(R(A^X), Y^0.operand)
        for i = 1, ..., s do
            if Y^0 has type hol_for_all  L.add(∀x.Z_i)
            if Y^0 has type hol_exists   L.add(∃x.Z_i)
            if Y^0 has type hol_lambda   L.add(λx.Z_i)
        [W_1, ..., W_s] = set_intersect(A^X, Y^0)
        for i = 1, ..., s do
            [W^0_1, ..., W^0_t] = set_subtract(W_i.operand, R(A^X))
            for j = 1, ..., t do
                if Y^0 has type hol_for_all  L.add(∀x.W^0_j)
                if Y^0 has type hol_exists   L.add(∃x.W^0_j)
                if Y^0 has type hol_lambda   L.add(λx.W^0_j)
    else if Y^0 has type hol_func_application
        let F be the function of Y^0 and [Y^0_1, ..., Y^0_N] be the arguments of Y^0
        [Z_1, ..., Z_s] = set_intersect_any_right(R(A^X), Y^0_N)
        for i = 1, ..., s do
            L.add(F(Y^0_1, ..., Y^0_{N-1}, Z_i))
        [W_1, ..., W_s] = set_intersect(A^X, Y^0)
        for i = 1, ..., s do
            let Q be the function of W_i and [S_1, ..., S_N] be the arguments of W_i
            [W^0_1, ..., W^0_t] = set_subtract(S_N, R(A^X))
            for j = 1, ..., t do
                L.add(Q(Y^0_1, ..., Y^0_{N-1}, W^0_j))
    else if Y^0 has type hol_true or hol_false or hol_variable
                or hol_constant or hol_string or hol_number
        L.add_all(set_intersect(A^X, Y^0))
    return L
```

2. In the second case, $x \in R(A^Y \setminus R(A^X))$, which means there is a subtree $b \in S_R(x)$ such that $b \in A^Y \setminus R(A^X)$. This implies that $x \in R(A^Y)$, and that there is a subtree $a \in S_R(b)$ such that $a \in A^X$. By definition, a is also a member of $S_R(x)$, and so $x \in R(A^X)$.

We conclude that $R(A^X \setminus R(A^Y)) \cup R(A^Y \setminus R(A^X)) \quad R(A^X) \setminus R(A^Y)$. By double containment, the two sets are equivalent.

SET DIFFERENCE: Observe that the helper function `set_intersect_any_right` relies on the set difference operation (via calls to `set_subtract`). Our algorithm for computing the set difference is shown in algorithm 16.

Just as with `set_intersect`, `set_subtract` also relies on a helper function `set_subtract_any_right` to handle the case where Y has type `hol_any_right`. This helper function is shown in algorithm 17.

The correctness of `set_subtract` in the case where both X and Y have type `hol_any_right` (lines 5-16 in algorithm 16) relies on the following observation: Let X and Y be written $X = R(A^X) \cap (B_1^X \cup \cdots \cup B_n^X)$ and $Y = R(A^Y) \cap (B_1^Y \cup \cdots \cup B_n^Y)$. Then:

$$
\begin{aligned}
&\left(R(A^X) \setminus \bigcup_{i=1}^{n} B_i^X\right) \setminus \left(R(A^Y) \setminus \bigcup_{i=1}^{m} B_i^Y\right), && (102)\\
&= R(A^X) \setminus \bigcap_{i=1}^{n} \overline{B}_i^X \setminus \left(R(A^Y) \setminus \bigcap_{i=1}^{m} \overline{B}_i^Y\right),\\
&= R(A^X) \setminus \bigcap_{i=1}^{n} \overline{B}_i^X \setminus \left(\overline{R(A^Y)} \cup \bigcup_{i=1}^{m} B_i^Y\right),\\
&= \left(R(A^X) \setminus \overline{R(A^Y)} \setminus \bigcap_{i=1}^{n} \overline{B}_i^X\right) \cup \bigcup_{i=1}^{m} \left(R(A^X) \setminus B_i^Y \setminus \bigcap_{j=1}^{n} \overline{B}_j^X\right),\\
&= \left(R(A^X) \setminus \left(R(A^Y) \cup \bigcup_{i=1}^{n} B_i^X\right)\right) \cup \bigcup_{i=1}^{m} (R(A^X) \setminus B_i^Y) \setminus \bigcup_{j=1}^{n} B_j^X.
\end{aligned}
$$

SET SUBSET: The last remaining set operation to complete the above algorithms is *set subset*. This operation is required by `set_intersect_any_right` (lines 12 and 15 in algorithm 15) and `set_subtract` (lines 7 and 18 in algorithm 16). To compute whether a set A is a subset of $B_1 \cup \cdots \cup B_n$, we rely on the fact that $A \quad (B_1 \cup \cdots \cup B_n)$ if and only if $A \cap (B_1 \cup \cdots \cup B_n) = A \cap B_1 \cap \cdots \cap B_n = ?$, and so we can apply `set_subtract` to implement the set subset operation for most inputs of A and B. However, this approach is insufficient when A has type `hol_any_right`, since `set_subtract` invokes the set subset operation on lines 7 and 18 in algorithm 16. To develop an algorithm to compute the subset operation in the case where A has type `hol_any_right`, we first prove a number of useful necessary and sufficient conditions for a set to be a subset of a union of sets.

Algorithm 16: Pseudocode to compute the set difference of two logical form sets, each represented by the `hol_term` data structure.

```
function set_subtract(hol_term X, hol_term Y)
    L is an empty list
    if X has type hol_any_right
        let R(A^X) n (B_1^X [ ::: [ B_n^X) be the set represented by X
        if Y has type hol_any_right
            let R(A^Y) n (B_1^Y [ ::: [ B_m^Y) be the set represented by Y
            if R(A^X) * (B_1^X [ ::: [ B_n^X [ R(A^Y))
                L.add(R(A^X) n (B_1^X [ ::: [ B_n^X [ R(A^Y)))
            for i = 1, :::, m do
                K = set_intersect_any_right(R(A^X), B_i^Y)
                for j = 1, :::, n do
                    K is an empty list
                    for each K^0 in K do L .add_all(subtract(K^0, B_j^X))
                    set K = K
                L.add_all(K)
            return L
        else
            if R(A^X)  B_1^X [ ::: [ B_n^X [ Y  return empty list
            else return R(A^X) n (B_1^X [ ::: [ B_n^X [ Y)
    else if Y has type hol_any_right
        return set_subtract_any_right(X, Y)
    else if X and Y have different types  return X
    if X has type hol_not
        [Z_1, :::, Z_n] = set_subtract(X.operand, Y.operand)
        for i = 1, :::, n do L.add(: Z_i)
    else if X has type hol_if_then
        [Z_1^L, :::, Z_n^L] = set_subtract(X.antecedent, Y.antecedent)
        let X^R be X.consequent
        for i = 1, :::, n do L.add(Z_i^L ! X_R)
        [Z_1^L, :::, Z_n^L] = set_intersect(X.antecedent, Y.antecedent)
        [Z_1^R, :::, Z_m^R] = set_subtract(X.consequent, Y.consequent)
        for i = 1, :::, n do
            for j = 1, :::, m do L.add(Z_i^L ! Z_j^R)
    else if X has type hol_equals
        [Z_1^L, :::, Z_n^L] = set_subtract(X.left, Y.left)
        let X^R be X.right
        for i = 1, :::, n do L.add(Z_i^L = X_R)
        [Z_1^L, :::, Z_n^L] = set_intersect(X.left, Y.left)
        [Z_1^R, :::, Z_m^R] = set_subtract(X.right, Y.right)
        for i = 1, :::, n do
            for j = 1, :::, m do L.add(Z_i^L = Z_j^R)
    else if X has type hol_and or hol_or
        let [X_1, :::, X_N] be X.operands
        for i = 1, :::, N - 1 do
            [Z_1^i, :::, Z_{n_i}^i] = set_intersect(X.operands[i], Y.operands[i])
        for i = 1, :::, N do
            [W_1^i, :::, W_{m_i}^i] = set_subtract(X.operands[i], Y.operands[i])
            for f(k_1, :::, k_i) : k_i 2 f1, :::, m_ig and k_j 2 f1, :::, n_jg for j < ig do
                if X has type hol_and
                    L.add(Z_{k_1}^1 ^ ::: ^ Z_{k_{i-1}}^{i-1} ^ W_{k_i}^i ^ X_{i+1} ^ ::: ^ X_N)
                else if X has type hol_or
                    L.add(Z_{k_1}^1 _ ::: _ Z_{k_{i-1}}^{i-1} _ W_{k_i}^i _ X_{i+1} _ ::: _ X_N)
```

Algorithm 16: (continued)

```
else if X has type hol_for_all or hol_exists or hol_lambda
    if X.operands.variable ≠ Y.operands.variable return X
    [Z_1, ..., Z_n] = set_subtract(X.operand, Y.operand)
    for i = 1, ..., n do
        let x be X.operands.variable
        if X has type hol_for_all  L.add(∀x.Z_i)
        if X has type hol_exists  L.add(∃x.Z_i)
        if X has type hol_lambda  L.add( x.Z_i)
else if X has type hol_func_application
    let [X_1, ..., X_N] be X.arguments
    [Z^0_1, ..., Z^0_{n_0}] = set_intersect(X.function, Y.function)
    [W^0_1, ..., W^0_{m_0}] = set_subtract(X.function, Y.function)
    for i = 1, ..., m_0 do L.add(W^0_i(X_1, ..., X_N))
    for i = 1, ..., N - 1 do
        [Z^i_1, ..., Z^i_{n_i}] = set_intersect(X.operands[i], Y.operands[i])
    for i = 1, ..., N do
        [W^i_1, ..., W^i_{m_i}] = set_subtract(X.operands[i], Y.operands[i])
        for f(k_0, ..., k_i) : k_i ∈ f1, ..., m_ig and k_j ∈ f1, ..., n_jg for j < ig do
            L.add(Z^0_{k_0}(Z^1_{k_1}, ..., Z^{i-1}_{k_{i-1}}, W^i_{k_i}, X_{i+1}, ..., X_N))
else if X has type hol_true or hol_false  return empty list
else if X has type hol_variable and X.variable ≠ Y.variable
    L.add(X)
else if X has type hol_constant and X.constant ≠ Y.constant
    L.add(X)
else if X has type hol_string and X.str ≠ Y.str
    L.add(X)
else if X has type hol_number and X.num ≠ Y.num
    L.add(X)
return L
```

Lemma 1. *For any logical form sets* A, B_1, ..., B_n, *each represented by the* `hol_term` *data structure, if every* B_i *of type* `hol_any_right` *has no* `excluded` *sets, then*

$$R(A) \quad B_1 \mid \cdots \mid B_n \quad \textit{if and only if} \quad R(A) \quad \bigcup_{i \in I} B_i,$$

where I , fi : B_i *has type* `hol_any_right`g *is the set of indices* i *such that* B_i *is of type* `hol_any_right`.

Proof. The *if* direction is true for any sets, since $\bigcup_{i \in I} B_i \quad \bigcup_{i=1}^{n} B_i$.

To show the *only if* direction, suppose to the contrary that R(A) * $\bigcup_{i \in I} B_i$, and so there exists a tree t ∈ R(A) such that t ∉ $\bigcup_{i \in I} B_i$. We will construct a new logical form t such that t ∈ R(A) but t ∉ $\bigcup_{i=1}^{n} B_i$, which would be contradiction. Inspect each logical form L(B_i) that has type `hol_for_all`, and let x_i be the declared variable. There exists a new variable x_k where $k > x_i$ for all i, which is undeclared by L(B_i). Construct a new logical form t = ∀x_k.t, where the root node of the expression tree has only one child t. Since t ∈ R(A),

Algorithm 17: Helper function to compute the set difference of two logical form sets where Y has type `hol_any_right`.

```
   /* precondition:  X does not have type hol_any_right  */
1  function set_subtract_any_right(hol_term X, hol_any_right Y)
2      let R(A) n (B_1 [ ::: [ B_n) be the set represented by Y
3      [Z_1, :::, Z_m] = set_subtract(X, A)
4      L is an empty list
5      if X has type hol_not
6          [W_1, :::, W_s] = set_subtract_any_right(X.operand, Y)
7          for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
8              [Z_1^0, :::, Z_t^0] = set_intersect(Z_i.operand, W_j)
9              for k = 1, :::, t do L.add( : Z_k^0)
10     else if X has type hol_if_then
11         [W_1, :::, W_s] = set_subtract_any_right(X.consequent, Y)
12         for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
13             let Z_L be Z_i.antecedent
14             [Z_1^0, :::, Z_t^0] = set_intersect(Z_i.consequent, W_j)
15             for k = 1, :::, t do L.add(Z_L ! Z_k^0)
16     else if X has type hol_equals
17         [W_1, :::, W_s] = set_subtract_any_right(X.right, Y)
18         for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
19             let Z_L be Z_i.left
20             [Z_1^0, :::, Z_t^0] = set_intersect(Z_i.right, W_j)
21             for k = 1, :::, t do L.add(Z_L = Z_k^0)
22     else if X has type hol_and or hol_or
23         let N be X.operands.length
24         [W_1, :::, W_s] = set_subtract_any_right(X.operands[N-1], Y)
25         for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
26             let [U_1, :::, U_N] be Z_i.operands
27             [Z_1^0, :::, Z_t^0] = set_intersect(U_N, W_j)
28             for k = 1, :::, t do
29                 if X has type hol_and L.add(U_1 ^ ::: ^ U_N-1 ^ Z_k^0)
30                 if X has type hol_or L.add(U_1 _ ::: _ U_N-1 _ Z_k^0)
31     else if X has type hol_for_all or hol_exists or hol_lambda
32         [W_1, :::, W_s] = set_subtract_any_right(X.operand, Y)
33         for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
34             let x be the variable of Z_i
35             [Z_1^0, :::, Z_t^0] = set_intersect(Z_i.operand, W_j)
36             for k = 1, :::, t do
37                 if X has type hol_for_all L.add(8x.Z_k^0)
38                 if X has type hol_exists L.add(9x.Z_k^0)
39                 if X has type hol_lambda L.add( x.Z_k^0)
40     else if X has type hol_func_application
41         let N be X.arguments.length
42         [W_1, :::, W_s] = set_subtract_any_right(X.arguments[N-1], Y)
43         for f(i, j) : i 2 f1, :::, mg and j 2 f1, :::, sgg do
44             let F be Z_i.function and let [U_1, :::, U_N] be Z_i.arguments
45             [Z_1^0, :::, Z_t^0] = set_intersect(U_N, W_j)
46             for k = 1, :::, t do L.add(F(U_1, :::, U_N-1, Z_k^0))
47     else if X has type hol_true or hol_false or hol_variable
                   or hol_constant or hol_string or hol_number
48         for i = 1, :::, m do L.add(Z_i)
49     return L
```

we have $t \in R(A)$. But since $t \notin L(B_i)$ for any i, $t \in B_i$ for any i. As such, $t \in B_1 \cap \bigcup_{i=1}^{n} B_i$, which is a contradiction, and so we conclude that $R(A) \subseteq \bigcup_{i \in I} B_i$.

We generalize this result to the case where we remove the constraint that all B_i of type `hol_any_right` have no `excluded` sets.

Thm 3. *Let* $A, B_1, \ldots, B_n$ *be logical form sets, each represented by the* `hol_term` *data structure, and the* B_i *with type* `hol_any_right` *are written* $B_i = R(U_i) \cap (V_{i1} \cap \ldots \cap Y_{im_i})$. *Let* $I, f_i : B_i$ *has type* `hol_any_right` *is the set of indices* i *such that* B_i *is of type* `hol_any_right`. *Define* $X_i, R(U_i)$ *and* $Y_i, V_{i1} \cap \ldots \cap Y_{im_i}$ *for all* $i \in I$, *and* X_i, B_i *and* $Y_i, ?$ *for all* $i \notin I$.

$$R(A) \subseteq \bigcup_{i=1}^{n} B_i \quad \textit{if and only if}$$

$$R(A) \subseteq \bigcup_{i \in I} X_i \textit{ and } \forall f \in F, \left(R(A) \setminus \bigcup_{i \notin I} X_i\right) \setminus \bigcap_{i \in I} f_i = ?,$$

where $F, f f : f_i \in f X_i, \overline{Y}_i g$ *where* $i \in I$ *and* $: \forall i, f_i = X_i g$ *is the set of all sequences where each sequence* f *is defined over the indices* $i \in I$, *and each element* f_i *is either* X_i *or* $\overline{Y}_i$, *excluding the sequence containing all* X_i.

Proof. **IF:** In the *if* direction, since $R(A) \subseteq \bigcup_{i \in I} X_i$, then by lemma 1, $R(A) \subseteq \bigcup_{i=1}^{n} X_i$. Next, we can re-write the union

$$\begin{aligned}
\bigcup_{i=1}^{n} B_i &= \bigcup_{i=1}^{n} X_i \cap Y_i = \bigcup_{i=1}^{n} X_i \setminus \overline{Y}_i \\
&= \bigcup_{i=1}^{n} X_i \setminus \bigcap_{f \in F} \left(\bigcup_{i \in I} \overline{f}_i \cup \bigcup_{i \notin I} X_i \right) \\
&= \bigcup_{i=1}^{n} X_i \setminus \bigcup_{f \in F} \left(\bigcap_{i \in I} f_i \setminus \bigcap_{i \notin I} \overline{X}_i \right).
\end{aligned}$$

Since for all $f \in F$, $(R(A) \cap \bigcup_{i \notin I} X_i) \setminus \bigcap_{i \in I} f_i = ?$, the $R(A)$ is disjoint from the subtracted set, and therefore $R(A) \subseteq \bigcup_{i=1}^{n} B_i$.

ONLY IF: Since $R(A) \subseteq \bigcup_{i=1}^{n} B_i$, it must be the case that $R(A) \subseteq \bigcup_{i=1}^{n} X_i$ and $R(A) \setminus \bigcup_{f \in F} (\bigcap_{i \in I} f_i \setminus \bigcap_{i \notin I} \overline{X}_i) = ?$, which implies

$$R(A) \setminus \bigcup_{f \in F} \left(\bigcap_{i \in I} f_i \setminus \bigcap_{i \notin I} \overline{X}_i \right) = ?,$$

$$\forall f \in F, R(A) \setminus \bigcap_{i \in I} f_i \setminus \bigcap_{i \notin I} \overline{X}_i = ?,$$

$$\forall f \in F, \left(R(A) \setminus \bigcup_{i \notin I} X_i\right) \setminus \bigcap_{i \in I} f_i = ?.$$

Also since $R(A) \subseteq \bigcup_{i=1}^{n} X_i$, then by lemma 1, $R(A) \subseteq \bigcup_{i \in I} X_i$.

Lemma 2. *For any sets* A *and* B, R(A) R(B) *if and only if* A R(B).

Proof. **IF:** We wish to show that if A R(B), then R(A) R(B). Take any element of R(A), t 2 R(A). By definition of R( ), there exists a right subtree t' 2 $S_R(t)$ such that t' 2 A. Thus, t' 2 R(B) and so there exists a right subtree t'' 2 $S_R(t')$ such that t'' 2 B. But since $S_R(t')$ $S_R(t)$, t is also in R(B). We conclude that R(A) R(B).

ONLY IF: Since A R(A) and R(A) R(B), A R(B).

Theorem 3 provides us with a way to compute whether a logical form set R(A) is a subset of the union of logical form sets $B_1 [\cdots [B_n$, where R(A) has type `hol_any_right` and no `excluded` sets. The procedure is as follows: Let I , fi : B_i has type `hol_any_right`g be the set of indices i such that B_i is of type `hol_any_right`, and for these i 2 I, B_i can be written B_i = R(U_i) n ($V_{i1} [\cdots [V_{i m_i}$).

1. Check whether R(A) $\bigcup_{i 2 I}$ R(U_i). By lemma 2, this is equivalent to checking A $\bigcup_{i 2 I}$ R(U_i). If not, return *false*.

2. Let F , ff : f_i 2 fR(U_i), n ($V_{i1} [\cdots [V_{i m_i}$)g where i 2 I and :8i, f_i = X_ig be the set of all sequences where each sequence f is defined over the indices i 2 I, and each element f_i is either X_i or $\overline{Y}_i$, excluding the sequence containing all X_i. For every f 2 F, check whether (R(A) n $\bigcup_{i 2 I}$ X_i) \ $\bigcap_{i 2 I}$ f_i = ?. If not, return *false*.

3. Otherwise, return *true*.

Theorem 3 cannot be extended to the case where the left-hand side set R(A) is allowed to be any `hol_term`. One counter-example is: the logical form set a ^ R(b), where a and b are constants, is a subset of the union of a ^ (a _ b) and R(a) ^ (R(b) n (a _ b)). But at the same time, a ^ R(b) is not a subset of a ^ (a _ b) alone, and a ^ R(b) is not a subset of R(a) ^ (R(b) n (a _ b)) alone. Incidentally, we never run into this case in any of our experiments, which suggests that under certain conditions which are met during our experiments, it may be the case that A $B_1 [\cdots [B_n$ if and only if A B_i for some i. We leave it to future work to identify these conditions if they exist.

Note that the worst-case computational complexity of the above algorithms for computing set operations is very high. In the subset operation above, the sequence F has length 2^k where k is the number of sets B_i that have type `hol_any_right`. In `set_intersect` (algorithm 14), in the case where the input set X has type `hol_and`, `hol_or`, or `hol_func_application`, on lines 26 and 43, the algorithm iterates over tuples of a Cartesian product, which can be very large. For instance, if X has type `hol_and` with N operands, and the recursive calls to `set_intersect` for each operand returned M sets, then the loop on line 26 would iterate over M^N tuples, and the function could return a list of M^N sets. However, we find that in practice, for the vast

majority of inputs during our experiments, the set operations return a single set (i.e. M = 1), which avoids the exponential blowup. In our implementation, we made an effort to avoid creating new sets unnecessarily. For example, if `set_intersect` returns either of its input sets (X or Y), instead of returning a copy, our implementation returns a pointer to the set. This enables optimizations throughout the code where we can compare whether two sets are equivalent first by checking whether their pointers are the same. Another optimization that we use is for commonly-used sets, such as , we define a global variable that can be re-used (e.g. `HOL_ANY` for) to avoid the overhead of creating the structure each time we need it. We do the same for other common logical forms, such as >, ?, the number 0, etc.

In addition, observe that the above representation for sets of logical forms is *closed* under the set intersection and difference operations: For any input logical form sets, represented by `hol_term`, their intersection and difference is a union of sets, each representable by `hol_term`. This is a desirable property for a data structure for sets of logical forms. However, we will show later that this property is not necessary, so long as the semantic transformation functions in the grammar avoid set operations that would produce logical forms that cannot be represented by the data structure.

CONJUNCTIONS AND DISJUNCTIONS WITH UNFIXED LENGTH: Many semantic transformation functions in our grammar operate on conjunctions and disjunctions of any length. For example, the transformation function `select_left_conjunct` selects two conjuncts from the head scope of the input logical form: (1) the left-most operand, and (2) the operand declaring the type of the head variable. Given the input logical form:

```
∃b(book(b)
    ^ ∃w(arg1(w)=me ^ write(w) ^ past(w) ^ arg2(w)=b))),
```

which represents the meaning of "I wrote a book," `select_left_conjunct` produces the output logical form:

```
∃w(write(w) ^ arg1(w)=me)).
```

The head scope in the above input logical form is ∃w(:::). But the function `select_left_conjunct` is not only limited to conjunctions with 4 operands as in the case above. As a result, it is impossible to properly implement the inverse of this transformation function using

the `hol_term` data structure defined thus far, since `hol_and` has a fixed length. To fix this, we introduce a new subtype of `hol_term`:

```
39 enum hol_array_operator
40 |  AND,
41 |  OR,
42 |  EITHER   /* AND or OR */
```

```
43 class hol_any_array extends hol_term
44 |  hol_array_operator operator
45 |  hol_term all
46 |  array<hol_term> left
47 |  array<hol_term> right
48 |  array<hol_term> any
```

This `hol_any_array` structure represents the set of all conjunctions or disjunctions that satisfy the following properties:

1. If the `operator` field is `AND`, the expression is a conjunction. If the `operator` field is `OR`, the expression is a disjunction.
2. Let a_i be the i^{th} operand of the conjunction/disjunction.
   a) Let A be the logical form set represented by the `all` field. For all i, $a_i \in A$.
   b) Let $(L_1, \ldots, L_n)$ be the array of logical form sets represented by the `left` field. For all $i = 1, \ldots, n$, $a_i \in L_i$.
   c) Let $(R_1, \ldots, R_m)$ be the array of logical form sets represented by the `right` field, and let N be the length of the conjunction/disjunction. For all $i = 1, \ldots, m$, $a_i \in R_{N+i-m}$.
   d) Let $(S_1, \ldots, S_k)$ be the array of logical form sets represented by the `any` field. There exists a j such that for all $i = 1, \ldots, k$, $a_i \in S_i$.

Stated plainly, the `left` field specifies the left-most operands of the conjunction/disjunction, the `right` field specifies the right-most operands of the conjunction/disjunction, and the `any` field specifies a sequence of operands that must appear somewhere within the conjunction/disjunction (in the same order).

As an example to illustrate the application of `hol_any_array`, the inverse of the `select_left_conjunct` function applied to the logical form

```
∃w(write(w) ^ arg1(w)=me))
```

is

```
R(∃w(any_array with operator = AND,
    left = [arg1(w)=me],  right = [],  any = [write(w)])).
```

This expression is the set of all logical forms with ∃w at the root, and whose child node is the set of all conjunctions where the left conjunct is `arg1(w)=me` and some conjunct is `write(w)`.

With this new subtype of `hol_term`, the set operations (in algorithms 14, 15, 16, and 17) need to be extended to handle the cases when their inputs have type `hol_any_array`. The set intersection operation for

Algorithm 18: Helper function for computing the intersection of two sets of logical forms, where X has type `hol_any_array`.

```
   /* precondition:  Y does not have type hol_any_right  */
 1 function set_intersect_any_array(hol_any_array X, hol_term Y)
 2     let L be an empty list
 3     if Y has type hol_any_array
 4         if X.operator = EITHER  let oper = Y.operator
 5         else if Y.operator = EITHER  let oper = X.operator
 6         else if X.operator = Y.operator  let oper = X.operator
 7         else return empty list
 8         let [S_1, ..., S_k] be X.any and let [S^0_1, ..., S^0_{k^0}] be Y.any
 9         if ∀i, ∀j ≥ 1, ..., k^0(S_{i+j-1}  S^0_j if i + j - 1 ≥ 1, ..., k, and
             X.all  S^0_j otherwise)
10             let [S_1, ..., S_k] = X.any      /* X.any is a subset of Y.any */
11         else if ∀i, ∀j ≥ 1, ..., k(S^0_{i+j-1}  S_j if i + j - 1 ≥ 1, ..., k^0, and
             Y.all  S_j otherwise)
12             let [S_1, ..., S_k] = Y.any      /* Y.any is a subset of X.any */
13         else  return unclosed operation error
14         let [L_1, ..., L_n] be X.left and let [L^0_1, ..., L^0_{n^0}] be Y.left
15         for i = 1, ..., min(n, n^0) do L_i = set_intersect(L_i, L^0_i)
16         for i = min(n, n^0), ..., n do L_i = set_intersect(L_i, Y.all)
17         for i = min(n, n^0), ..., n^0 do L_i = set_intersect(L^0_i, X.all)
18         let [R_1, ..., R_m] be X.right and let [R^0_1, ..., R^0_{m^0}] be Y.right
19         let M^+ = max(m, m^0) and M^- = min(m, m^0)
20         for i = 1, ..., M^- do R_{M^+ - i+1} = set_intersect(R_{m - i+1}, R^0_{m^0 - i+1})
21         for i = M^-, ..., m do R_{M^+ - i+1} = set_intersect(R_{m - i+1}, Y.all)
22         for i = M^-, ..., m^0 do R_{M^+ - i+1} = set_intersect(R^0_{m^0 - i+1}, Y.all)
23         [A_1, ..., A_r] = set_intersect(X.all, Y.all)
24         for i = 1, ..., k  do
25             let L^0 be an empty list
26             for j = 1, ..., r do L^0.add_all(set_intersect(S_i, A_j))
27             set S_i to be L^0
28         let N^+ = max(n, n^0)
29         for i = 1, ..., r do
30             for f(i_1, ..., i_{N^+}) : i_u ≥ 1, ..., j|L_u|g do
31                 for f(j_1, ..., i_{M^+}) : j_u ≥ 1, ..., j|R_u|g do
32                     for f(a_1, ..., a_k) : a_u ≥ 1, ..., j|S_u|g do
33                         L.add( new any_array with operator oper, all = A_i,
                             left = [L_{1 i_1}, ..., L_{N^+ i_{N^+}}], right = [R_{1 j_1}, ..., R_{M^+ j_{N^+}}], and
                             any = [S_{1 a_1}, ..., S_{k a_k}])
34     else if Y has type hol_and or hol_or
35         if X.operator = AND and Y does not have type hol_and  return empty list
36         if X.operator = OR and Y does not have type hol_or  return empty list
37         let [L_1, ..., L_n] be X.left
38         let [R_1, ..., R_m] be X.right
39         let [Y_1, ..., Y_N] be Y.operands
40         if N < max(n, m, k)  return empty list
41         for i = 1, ..., N do
42             Y_i = set_intersect(Y_i, X.all)
43             if i ≠ n
44                 let L^0 be an empty list
45                 for each U ≥ Y_i do L^0.add_all(set_intersect(U, L_i))
46                 set Y_i to be L^0
47             if i > N - m
48                 let L^0 be an empty list
49                 for each U ≥ Y_i do L^0.add_all(set_intersect(U, R_{i - N+m}))
50                 set Y_i to be L^0
```

Algorithm 18: (continued)

```
51    for f(i_1, ..., i_N) : i_j ∈ 1, ..., jY_i jgg do
52        let [S_1, ..., S_k] be X.any
53        for j = 1, ..., N - k + 1 do
54            for u = 1, ..., k do Y_{j+u-1} = set_intersect(Y_{(j+u-1), i_{j+u-1}}, S_u)
55            for f(u_j, ..., u_{j+k} : u_a ∈ 1, ..., jY_j jgg do
56                let L^0 be an empty list
57                for p = 1, ..., N do
58                    if j ≤ p < j + k  L^0_p = Y_{p u_p}
59                    else L^0_p = Y_{p i_p}
60                if Y has type hol_and L.add(L^0_1 ∧ ... ∧ L^0_N)
61                if Y has type hol_or L.add(L^0_1 ∨ ... ∨ L^0_N)
62    return L
```

Algorithm 19: The *if* statement that is added to `set_intersect_any_right` (algorithm 15) on line 87 to handle the case where Y^0 has type `hol_any_array`.

```
87    else if Y^0 has type hol_any_array
88        let [R_1, ..., R_m] be Y^0.right
89        if m ≠ 0  [U_1, ..., U_r] = set_intersect_any_right(R(A^X), R_m)
90        else [U_1, ..., U_r] = set_intersect_any_right(R(A^X), Y^0.all)
91        for each i ∈ 1, ..., r do
92            L.add( new hol_any_array identical to Y^0 except the last element of
                  right is U_i )
93        [V_1, ..., V_s] = set_intersect(A^X, Y^0)
94        for i = 1, ..., s do
95            if V_i has type hol_any_array
96                let [R^0_1, ..., R^0_m] be V_i.right
97                if m ≠ 0  [W_1, ..., W_t] = subtract_any_right(R^0_m, R(A^X))
98                else [W_1, ..., W_t] = subtract_any_right(V_i.all, R(A^X))
99                for j = 1, ..., t do
100                   L.add( new hol_any_array identical to V_i except the last element
                        of right is W_j )
101           else  /* V_i has type hol_and or hol_or */
102               let [R^0_1, ..., R^0_m] be V_i.operands
103               [W_1, ..., W_t] = subtract_any_right(R^0_m, R(A^X))
104               for j = 1, ..., t do
105                   if V_i has type hol_and  L.add(R^0_1 ∧ ... ∧ R^0_{m-1} ∧ W_j)
106                   if V_i has type hol_or  L.add(R^0_1 ∨ ... ∨ R^0_{m-1} ∨ W_j)
```

Algorithm 20: The *if* statement that is added to `set_subtract` (algorithm 16) on line 20 to handle the case where either X or Y has type `hol_any_array`.

```
20  else if X has type hol_any_array
21    if Y has type hol_any_array
22      if X.operator ∉ EITHER and Y.operator ∉ EITHER and
          X.operator ≠ Y.operator  return [X]
23      let [L_1, ..., L_n] be X.left and let [L_1^0, ..., L_n0^0] be Y.left
24      let [R_1, ..., R_m] be X.right and let [R_1^0, ..., R_m0^0] be Y.right
25      let [S_1, ..., S_k] be X.any and let [S_1^0, ..., S_k0^0] be Y.any
26      if X.all = Y.all and ∀i ∈ 1, ..., minfn, n^0_g, L_i = L_i^0 and
          ∀i ∈ fn, ..., n^0_g, X.all = L_i^0 and
          ∀i ∈ 1, ..., minfm, m^0_g, R_{m-i+1} = R^0_{m0-i+1} and
          ∀i ∈ fm, ..., m^0_g, X.all = R^0_{m0-i+1} and
          ∃i, ∀j ∈ 1, ..., k^0_g(S_{i+j-1} = S_j^0 if i + j - 1 ∈ 1, ..., k_g, and
          X.all = S_j^0 otherwise)
27        return empty list      /* X is a subset of Y */
28      if set_intersect(X, Y) is empty  return [X]
29      if X.all = Y.all and ∀i, X.all ∈ L_i^0 and ∀i, X.all ∈ R_i^0 and
          ∀i, X.all ∈ S_i^0
30        let L be an empty list
31        for i = 2, ..., maxfn^0, m^0, k^0_g - 1 do
32          if X.operator = EITHER or X.operator = AND
33            let L^0 = L_1^0 ^ ... ^ L_i^0 where L_j^0 = ∈ for each j
34            L.add_all(set_intersect(X, L^0))
35          if X.operator = EITHER or X.operator = OR
36            let L^0 = L_1^0 _ ... _ L_i^0 where L_j^0 = ∈ for each j
37            L.add_all(set_intersect(X, L^0))
38        return L
39      return unclosed operation error
40    else if Y has type hol_and or hol_or
41      if Y has type hol_and and X.operator ∉ AND  return [X]
42      if Y has type hol_or and X.operator ∉ OR  return [X]
43      if set_intersect(X, Y) is empty  return [X]
44      else  return unclosed operation error
45    else  return [X]
46  else if Y has type hol_any_array
47    if (X has type hol_and and Y.operator ∉ EITHER and Y.operator ∉ AND)
        or (X has type hol_or and Y.operator ∉ EITHER and Y.operator ∉ OR)
        or (X does not have type hol_and or hol_or)
48      return [X]
49    let [X_1, ..., X_N] be X.operands
50    let [L_1, ..., L_n] be Y.left
51    let [R_1, ..., R_m] be Y.right
52    let [S_1, ..., S_k] be Y.any
53    if ∀i ∈ 1, ..., N_g, X_i = Y.all and ∀i ∈ 1, ..., n_g, X_i = L_i and
        ∀i ∈ 1, ..., m_g, X_{N-i+1} = R_{m-i+1} and ∃i, ∀j ∈ 1, ..., k_g, X_{i+j} = S_j
54      return empty list
55    if set_intersect(X, Y) is empty  return [X]
56    return unclosed operation error
```

the case where either input set has type `hol_any_array` is shown in algorithm 18. Notice that on line 13, the algorithm throws an error indicating that the set operation is unclosed (i.e. it would produce a set that cannot be represented by `hol_term`). To avoid this error, we could modify the any field to be an array of an array, where each inner array must appear somewhere in the conjunction/disjunction. However, our implementation does not reach that point in the code, suggesting that the our grammar and/or transformation functions never call `set_intersect_any_array` with inputs such that the resulting intersection is unclosed.

In the `set_intersect_any_right` helper function (in algorithm 15), we need to add an *if* statement on line 87 to check for the case that Y^0 has type `hol_any_array`. This *if* block is shown in algorithm 19.

Algorithm 21: The *if* statement that is added to `set_subtract_any_right` (algorithm 17) on line 49 to handle the case that X has type `hol_any_array`.

```
49  else if X has type hol_any_array
50      let [R_1, ..., R_m] be X.right
51      if m ≠ 0  [W_1, ..., W_s] = set_subtract_any_right(R_m, Y)
52      else  [W_1, ..., W_s] = set_subtract_any_right(X.all, Y)
53      for f(i, j) : i ∈ 1, ..., m_g and j ∈ 1, ..., s_gg do
54          let [U_1, ..., U_{m^0}] be Z_i.right
55          if m^0 ≠ 0  [Z^0_1, ..., Z^0_t] = set_intersect(U_{m^0}, W_j)
56          else  [Z^0_1, ..., Z^0_t] = set_intersect(Z_i.all, W_j)
57          for k = 1, ..., t do
58              L.add( new hol_any_array identical to Z_i except the last element of
                  right is Z^0_k )
```

To extend `set_subtract` (algorithm 16) to handle the case where either input has type `hol_any_array`, we need to add an *if* statement on line 20. This *if* block is shown in algorithm 20.

Finally, we extend `set_subtract_any_right` (algorithm 17) to handle the case where the input X has type `hol_any_array` by adding an *if* statement on line 49. This *if* block is shown in algorithm 21.

SETS OF CONSTANTS: Another useful subtype of `hol_term` in our implementation is to represent sets of constants. While `hol_any_right` can be used to represent sets of constants, it is highly non-specific, and a set represented by `hol_any_right` contains many other logical forms that are not constants. Many semantic feature functions and transformation functions work with these constants. For example, the HDP hierarchies for the `N`, `V`, `ADJ`, `ADV` nonterminals are constructed using the constant value of the input logical form. That is, they use a feature function that when given a logical form A, returns A if A is a constant, and otherwise returns *null*. To properly implement the `get_feature`, `set_feature`, and `exclude_feature` for this feature function (as described in section 4.1.4). As another example, our grammar

has transformation functions that manipulate predicates that express the tense and aspect of the sentence, and these predicates are themselves constants. A data structure that provides an easy way to work with sets of constants is valuable. To this end, we introduce two new subtypes of `hol_term`:

```
49 class hol_any_constant
     extends hol_term
       /* length must be at least 2 */
50     array<int> constants
```

```
51 class hol_any_constant_except
     extends hol_term
       /* possibly empty */
52     array<int> excluded
```

Algorithm 22: Helper functions for computing the intersection of two sets of logical forms, where X has type `hol_any_constant` or `hol_any_constant_except`.

```
   /* precondition:  Y is not hol_any_right or hol_any_array  */
1  function set_intersect_any_constant(
                hol_any_constant X, hol_term Y)
2    if Y has type hol_any_constant
3      let C = fc : c 2 X.constants and c 2 Y.constantsg
4    else if Y has type hol_any_constant_except
5      let C = fc : c 2 X.constants and c 2 Y.excludedg
6    else if Y has type hol_constant
7      if Y.constant 2 X.constants  return Y
8      else return empty list
9    else return empty list
10   if C = ?  return empty list
11   else if C = fcg is a singleton  return c
12   return [new hol_any_constant with constants = C]
13 function set_intersect_any_constant_except(
                hol_any_constant_except X, hol_term Y)
14   if Y has type hol_any_constant
15     let C = fc : c 2 X.excluded and c 2 Y.constantsg
16     if C = ?  return empty list
17     else if C = fcg is a singleton  return c
18     return [new hol_any_constant with constants = C]
19   else if Y has type hol_any_constant_except
20     let C = fc : c 2 X.excluded or c 2 Y.excludedg
21     return [new hol_any_constant_except with excluded = C]
22   else if Y has type hol_constant
23     if Y.constant 2 X.excluded  return empty list
24     else return Y
25   else
26     return empty list
```

`hol_any_constant` represents a union of two or more constants, whereas `hol_any_constant_except` represents the set of all constants *except* zero or more constants.

As with `hol_any_array`, we extend the set operation algorithms to handle the new subtypes. The set intersection operation for the case where either input set has type `hol_any_constant` or `hol_any_constant_except` is shown in algorithm 22.

Algorithm 23: The *if* statement that is added to `set_intersect_any_right` (algorithm 15) on line 87 to handle the case where Y^0 has type `hol_any_constant` or `hol_any_constant_except`.

```
87  else if Y⁰ has type hol_any_constant
88  |  L.add_all(set_intersect_any_constant(Y⁰, A^X))
89  else if Y⁰ has type hol_any_constant_except
90  |  L.add_all(set_intersect_any_constant_except(Y⁰, A^X))
```

Algorithm 24: The *if* statement that is added to `set_subtract` (algorithm 16) on line 20 to handle the case where either X or Y has type `hol_any_constant` or `hol_any_constant_except`.

```
20  else if X has type hol_any_constant
21  |  if Y has type hol_any_constant
22  |  |  let C = fc : c 2 X.constants and c 2 Y.constantsg
23  |  else if Y has type hol_any_constant_except
24  |  |  let C = fc : c 2 X.constants and c 2 Y.excludedg
25  |  else if Y has type hol_constant
26  |  |  let C = fc : c 2 X.constants and c 6= Y.constantg
27  |  else return [X]
28  |  if C = ? return empty list
29  |  else if C = fcg is a singleton return c
30  |  return [new hol_any_constant with constants = C]
31  else if Y has type hol_any_constant
32  |  if X has type hol_any_constant_except
33  |  |  let C = fc : c 2 X.excluded or c 2 Y.constantsg
34  |  |  return [new hol_any_constant_except with excluded = C]
35  |  else if X has type hol_constant
36  |  |  if X.constant 2 Y.constants return empty list
37  |  |  else return [X]
38  |  else return [X]
39  else if X has type hol_any_constant_except
40  |  if Y has type hol_any_constant_except
41  |  |  let C = fc : c 2 X.excluded and c 2 Y.excludedg
42  |  |  if C = ? return empty list
43  |  |  else if C = fcg is a singleton return c
44  |  |  return [new hol_any_constant with constants = C]
45  |  else if Y has type hol_constant
46  |  |  let C = fc : c 2 X.excluded and c = Y.constantg
47  |  |  return [new hol_any_constant_except with excluded = C]
48  |  else return [X]
49  else if Y has type hol_any_constant_except
50  |  if X has type hol_constant
51  |  |  if X.constant 2 Y.excluded return [X]
52  |  |  else return empty list
53  |  else return [X]
```

In the `set_intersect_any_right` helper function (in algorithm 15), we add another *if* statement on line 87 to check for the case that Y^0 has type `hol_any_constant` or `hol_any_constant_except`. These *if* blocks are shown in algorithm 23.

Algorithm 25: The *if* statement that is added to `set_subtract_any_right` (algorithm 17) on line 49 to handle the case that X has type `hol_any_constant` or `hol_any_constant_excluded`.

```
49  else if X has type hol_any_constant or hol_any_constant_excluded
50      for i = 1,:::,m do L.add(Z_i)
```

To extend `set_subtract` (algorithm 16) to handle the case where either input has type `hol_any_constant` or `hol_any_constant_except`, we add another *if* statement on line 20. This *if* block is shown in algorithm 24.

Finally, we extend `set_subtract_any_right` (algorithm 17) to handle the case where the input X has type `hol_any_constant` or `hol_any_constant_excluded` by adding another *if* statement on line 49. This *if* block is shown in algorithm 25.

4.5.3 *Training*

In order to use this new grammar for parsing, we first need to infer the posterior distribution of the production rules of the new grammar, as well as induce the preterminal production rules. Since PWL uses an HDP for the conditional distribution of selecting production rules, inferring the posterior of the production rules is equivalent to computing MCMC samples of the seating assignments in the Chinese restaurant franchise corresponding to each HDP (see section 4.1.1; and recall that PWL only keeps the last sample $N_{\text{samples}} = 1$). To infer the posterior on the production rules, we construct a small *seed training set* consisting of 44 labeled sentences, 33 nouns, 42 adjectives, and 14 verbs. The seed training set is available at `github.com/asaparov/PWL/blob/main/seed_training_set.txt`. One example from the seed training set is shown in figure 18. We wrote and labeled these sentences by hand, which are largely from the domain of astronomy, with the aim to cover a diverse range of English syntactic constructions. This small training set was sufficient thanks to the statistical efficiency of the model, and we found that a smaller handful of "prototypical" sentences was good enough for robust and accurate parsing, in the sense that each sentence exhibits some syntactic structure that the other examples do not exhibit.

To facilitate debugging of the semantic transformation functions, each sentence is labeled not only with its logical form but also its full derivation tree. Derivation tree labels are not necessary, since section 4.3.1 details a method to sample the derivation trees in case any or all

Sentence: "Which inner planet has the highest mass?"

Logical form:
```
z. X(X= x( i(inner(i)   arg1_of(x)=i)   planet(x))
   X(z)    f((f= x. v. m( y(value(y)   arg2(y)=v   arg1_of(m)=y)
   mass(m)    h(arg1(h)=x   has(h)   present(h)   arg2(h)=m)))
    g(greatest(f)(g)   arg1(g)=X   arg2(g)=z)))
```
(i.e. what is the value of z such that there exists a set of inner planets X, z is a member of X, and there exists a function f that returns the mass of its input, such that z maximizes f over the set X)

Derivation tree:

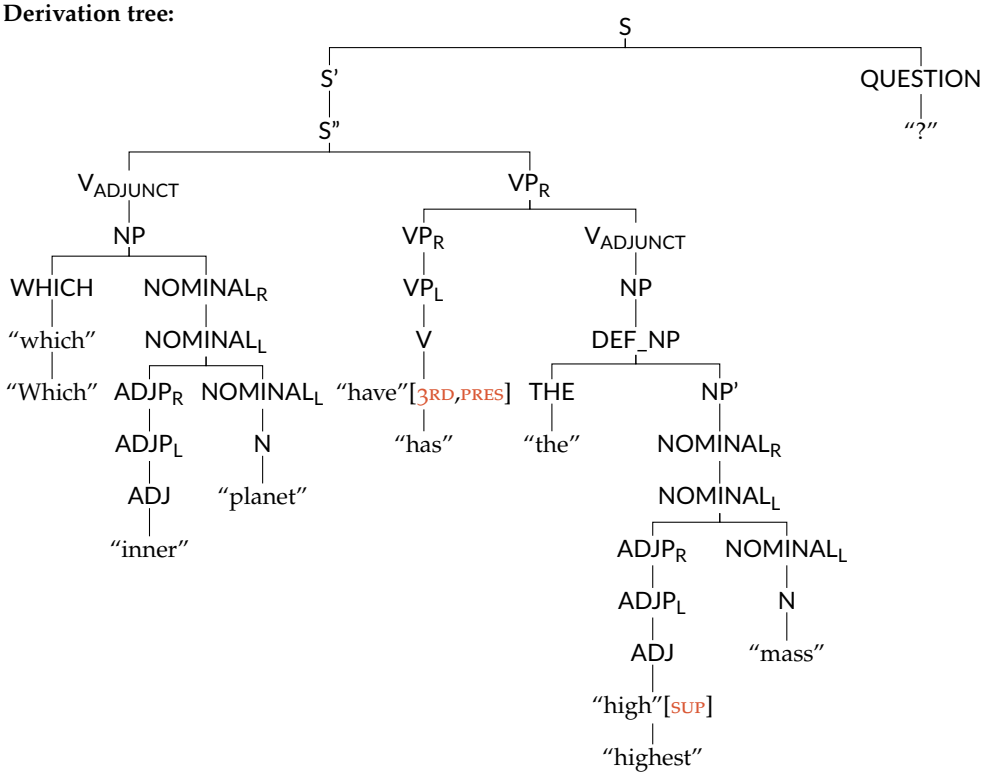


Figure 18: An example from the seed training set of PWL, labeled with the logical form and derivation tree (i.e. syntax tree). This example helps to train the parser in PWL. Note that the derivation tree label is not strictly necessary, as the training algorithm in section 4.3.1 can infer the latent derivation trees from sentences with logical form labels. In the above example derivation tree, 3RD is a morphological flag that indicates the third person, PRES indicates present tense, and SUP indicates superlative. Semantic transformation functions are omitted for brevity.

Algorithm 26: Pseudocode to check whether a given derivation tree t is parseable if the initial set of logical forms is X.

```
function is_parseable(logical form set X, expected derivation tree t)
    n is the root of the derivation tree t
    x is the logical form at n
    if n has a single child node that is a terminal w
        /* if using a morphology model, make sure that w is a valid
           morphological parse at this position */
        if x 2 X return ?
        X = is_rule_parseable(A ! w, X, x)
        if X = ? return ?
        set X = X
    else
        A ! B1:f1 ::: BK:fK is the production rule at n
        for i 2 1, ::: , K do
            Xi = fi(X)
            if Xi = ?
                return ? /* the output of the function fi is incorrect */
            xi = fi(fxg)
            if xi = ?
                return ? /* the output of the function fi is incorrect */
            c is the i^th child node of n
            if h^c_x(Xi) = - 1
                return ? /* the semantic prior heuristic is incorrect */
            ti is the subderivation of t rooted at the i^th child node of t
            Xi = is_parseable(Xi, ti)
            if Xi = ?
                return ?
            /* the operation X \ fi^-1(Xi) can return a union of sets */
            let Y1 [ ::: [ Yp be the output of X \ fi^-1(Xi)
            /* find the Yj that contains the correct logical form */
            j is the index such that x 2 Yj
            if there is no j such that x 2 Yj
                return ? /* the output of the function fi^-1 is incorrect */
            if x \ fi^-1(xi) = ?
                return ? /* the output of the function fi^-1 is incorrect */
            else if h^n_x(Yj) > h^n_x(X) or h^n_x(Yj) > h^c_x(Xi) or h^n_x(Yj) = 1
                return ? /* the semantic prior heuristic is incorrect */
            set X = Yj
        X = is_rule_parseable(A ! B1:f1 ::: BK:fK, X, x)
        if X = ? return ?
        set X = X
    return X
```

Algorithm 27: Recall that in the HDP model of section 4.1.4, each leaf node in the hierarchy corresponds to a set of logical forms (as defined by the functions `get_feature` and `set_feature`). Let S_x be the set of logical forms that corresponds to a leaf node in the HDP hierarchy such that $x \in S$. Given a logical form x, logical form set X, and production rule r, this algorithm finds the appropriate HDP hierarchy and then computes $S_x \setminus X$.

```
1 function is_rule_parseable(production rule r,
                             logical form set X,
                             logical form x)
2     A is the left-hand nonterminal symbol of r
3     f_1, ..., f_d are the semantic feature functions in the HDP for A
4     for i = 1, ..., d do
5         X = set_feature(f_i, X, get_feature(f_i, x))
6         if X = ? return ?
7         set X = X
8     return X
```

of them are latent/unknown. Derivation tree labels were not provided in the experiments on GeoQuery and Jobs in section 4.4.

We implemented a function `is_parseable` which would essentially simulate the parser's steps in the search for a given derivation tree. A bug in the implementation of a semantic transformation function or its inverse would cause `is_parseable` to return false and report at which step the failure occurred. Similarly the function can help to find bugs in the grammar itself. If a production rule is missing or incorrectly written, this function will report the an error at the incorrect rule. The function also checks for errors in morphological parsing or the heuristic upper bound for the semantic prior. The function is shown in algorithm 26. This function is first invoked with X being the set of all logical forms. In principle, it may return a set of logical forms rather than a singleton, which would indicate that the semantic transformation functions and feature functions are insufficient to partition the set X into a subset that only contains the ground truth logical form. In this case, the parser also would not return a singleton logical form. Rather, it would return a set of logical forms that contains the ground truth logical form. This provides a good indication to modify the semantic transformation functions and feature functions so that the parser can correctly find the singleton set containing the ground truth logical form.

Recall that the parser, given a production rule r and logical form set X, iterates over the logical form sets that maximize the likelihood $p(r \mid x \in X, t)$ (on line 28 in algorithm 26). To check for the correctness of this step, `is_parseable` calls a helper function `is_rule_parseable` (on lines 6 and 34). The implementation of this function depends on the model for choosing production rules. As PWL uses the HDP model as discussed in section 4.1.4, an implementation is provided in algorithm 27.

4.6 RELATED WORK

Our grammar formalism can be related to synchronous CFGs (SCFGs) (Aho and Ullman, 1972), where the semantics and syntax are generated simultaneously. However, instead of modeling the joint probability of the logical form and natural language utterance $p(x, y)$, we model the factorized probability $p(x)p(y \mid x)$, where the logical form x may have its own complex prior distribution $p(x)$. Modeling each component in isolation provides a cleaner division between syntax and semantics, and one half of the model can be modified without affecting the other, and this is instrumental in PWM since in that model, the logical form is derived from a larger theory containing background knowledge. We used a CFG in the syntactic portion of our model (although our grammar is not context-free, due to the dependence on the logical form). Richer syntactic formalisms such as combinatory categorial grammar (Steedman, 1997) or head-driven phrase structure grammar (Proudian and Pollard, 1985) could replace the syntactic component in our framework and may provide a more uniform analysis across languages. Our model is similar to lexical functional grammar (LFG) (Kaplan and Bresnan, 1995), where f-structures are replaced with logical forms. Nothing in our model precludes incorporating syntactic information like f-structures into the logical form, and as such, LFG is realized in our framework. In fact, including a model of morphology in our grammar furthers its similarity to LFG. Our approach can be used to define new generative models of these grammatical formalisms. We implemented our method with a particular semantic formalism, but the grammatical model is agnostic to the choice of semantic formalism or the language. As in some previous parsers, our parsing problem can be related to the problem of finding shortest paths in hypergraphs using A* search (Gallo, Longo, and Pallottino, 1993; Klein and Manning, 2001, 2003; Pauls and Klein, 2009; Pauls, Klein, and Quirk, 2010).

4.7 FUTURE WORK

There is significant room for future work and exploration in the subject presented in this chapter. In this section, we discuss shortcomings of various aspects of our approach, and give suggestions for how to overcome them.

4.7.1 *Shortcomings of the grammatical framework*

The performance of our parser and generator depend heavily on the production rules of the grammar. Although the preterminal production rules are induced during training, we had to specify the other production rules by hand. While this does give us a great deal of control over the grammar, and enables us to incorporate prior knowl-

edge about the English language into the grammar, it is very time-consuming. It would be valuable to look into ways in which these production rules can be induced from data. Recall that every production rule in our grammar is annotated with semantic transformation functions. These functions are intimately tied with the semantic formalism and effectively implement a theory of formal semantics. It would also be valuable to explore whether these transformation functions can be learned as well. One promising direction would be to decompose the semantic transformation functions into a sequence of elementary "instructions." Each semantic transformation function could then be equivalently written as short programs in a simple programming language. We could then induce the semantic transformation functions by searching over the space of these short programs, perhaps by attempting to add or remove instructions, etc. However, it is not clear how much grammar induction would improve our current grammar for English. But such an approach would certainly help to learn grammar for other languages, about which we have much less knowledge. The statistical efficiency of our approach could greatly aid in natural language processing for low-resource languages, for which training data is very scarce.

During parsing, PWL uses an upper bound on the objective function (as defined in equations 87, 88, and 89) that takes into account syntactic information. While this works well enough for our purposes, it may be possible to further improve the performance of the parser by defining tighter upper bound, possibly by taking into account semantic information.

The language module of PWL assumes that the sentences are noiseless: there are no spelling or grammatical errors in the utterances. This assumption helps to simplify the problem and to focus the scope of the thesis more onto language understanding and reasoning. But real-world language is noisy, and thus further work to extend the language module to noisy settings is warranted. To properly handle grammatical errors, additional "incorrect" production rules must be added to the grammar, such as a rule where the grammatical number of the subject noun and the verb do not agree, or a rule where the subject is dropped entirely (and left to be inferred from context). Grammar induction could be used to learn these "incorrect" production rules. A possible way to handle spelling errors is to add another step to the generative process for the language module (as described in section 4.2.1). This extra step would take the correctly-spelled sentence as its input and create errors, such as insertions, deletions, or substitutions of characters. During inference, this process is inverted: Given the noisy sentence as input, the parser first needs to infer the correctly-spelled sentence (which is now latent), and then proceed with the parsing algorithm as described earlier in this chapter.

4.7.2 *Shortcomings of the grammar*

Our new grammar was designed to broadly cover English, but it does not currently support a number of core features of English, such as interrogative subordinate clauses, wh-movement, imperative mood, and others. However, it is not difficult to extend the grammar to handle these features by adding additional production rules. For example, wh-movement can be added by defining a new transformation function that searches for the scope (existentially-quantified subexpression) within the input logical form that references the interrogative variable (i.e. the variable `x` if the input logical form looks like `x(:::)`). This scope may be nested within the semantic head of the input logical form. If the function encounters any "barriers" in its search such as negation, it would return failure. These movement restrictions are well-studied in linguistics and are known as *island effects*, and the new transformation function can be implemented to enforce these known barriers to movement.

SHORTCOMINGS OF THE SEMANTIC REPRESENTATION: Both the Datalog representation of the GeoQuery and Jobs datasets and the new semantic formalism presented in section 4.5 are not able to correctly represent sentences with *intension* and *modality*. Consider the sentence "I seek a unicorn." In our new semantic formalism, the meaning of the sentence is represented as

```
∃u(unicorn(u) ∧ ∃s(arg1(s) = me
  ∧ seek(s) ∧ present(s) ∧ arg2(s) = u))).
```

This logical form declares the existence of an object with type `unicorn`. So according to our formalism, the statement "I seek a unicorn" implies "There is a unicorn." In formal semantics, there are a number of ways to address this problem. One such approach is to distinguish between the real world and the world containing all objects, both real and hypothetical (Hobbs, 1985). Quantifiers would quantify over both real and hypothetical objects by default. A new predicate would be introduced to declare that an object exists in the real world. In this approach, we could represent "I seek a unicorn" as:

```
∃u(unicorn(u) ∧ ∃s(arg1(s) = me
  ∧ seek(s) ∧ real(s) ∧ present(s) ∧ arg2(s) = u))).
```

Here, only the `seek` event is marked as real, whereas the instance of `unicorn` is not. Axioms can be added so that for most events `e`, if `e` is real, then the arguments of `e` are real. But importantly, for event types such as `seek`, `think`, `believe`, `want`, `is_capable`, etc, this property would not hold. Another way to handle intensionality and modality is modify the formal language itself, as in the approach of Montague (1973). The above is an example of a broader distinction between the

de re and *de dicto* readings of the sentence "I seek a unicorn." This distinction is clearer in the example "John believes someone is a spy." In the *de re* reading, John believes that a spy exists in the world, but in the *de dicto* reading, there exists a person who John believes to be a spy (Quine, 1956). For this thesis, we assume that sentences do not express uncertainty or possibility, which allowed us to sidestep the need to properly represent intensionality and modality in the formal language. Thus it would be valuable to extend the semantic formalism to distinguish between these readings.

In addition, our new semantic formalism is not able to identify all of the interpretations of clauses with multiple universal quantifiers, such as in "Three teachers graded 6 exams" (Scha, 1981). In the first reading, this sentence means that each of the three teachers graded six exams, for a total of up to 18 graded exams. In the second reading, there exists a set of three teachers and a set of six exams, where each teacher helped to grade every exam and each exam was graded by every teacher. In the third reading, again there exists a set of three teachers and a set of six exams, but every teacher graded at least one exam, and every exam was graded by at least one teacher. In principle, it is possible to represent all three readings in first- and higher-order logic, but our grammar does not currently allow our parser to produce the third reading. A new transformation function and/or production rule would enable to parsing of this reading.

4.7.3 *Modeling context*

The logical forms in PWM are assumed to be context-independent. Conditioned on the theory, they are independently and identically distributed. This assumption greatly simplifies the natural language that we need to be able to parse. While it helps to focus the scope of thesis, it is not representative of real-world language. In real language, the distribution of a sentence is highly dependent on the sentences that precede it, even when conditioned on the theory, which contains all of the background knowledge. For example, this assumption disallows inter-sentential coreference (e.g. pronouns that can refer to objects mentioned in other sentences). PWM also assumes that the universe of discourse does not vary, and so the sentence "All of the children are asleep" would mean that, literally, every child in the universe is sleeping. The more likely meaning of the sentence is that all of the children within the local area, such as the home or town, are sleeping. The definite article "the" often indicates the uniqueness of an object: "the tallest mountain" indicates that there is exactly one tallest mountain. However, this is not the case in the example: "A cat walked into the room. The cat purred." Here, "the cat" does not imply that there is exactly one cat in the universe. Rather, it means that the cat is unique in the context. The universe of discourse can change

across sentences (and sometimes even within sentences). Relaxing the assumption that logical forms are context-independent would enable our parser to correctly understand these example sentences.

To relax this assumption, PWM must be augmented with a model of context. One possible approach is to modify the generative process of the proofs, which is described in section 3.2.2. The model can be modified to generate axioms in the same order in which the corresponding phrases appear in the sentence. The distribution of these axioms would now depend on an additional "context" random variable. This context variable includes the current universe of discourse as well as recently-mentioned entities. With every generated axiom, the context variable is modified probabilistically, such as by adding an entity to the list of recently-mentioned entities, and/or by widening/narrowing the universe of discourse. For example, if c_i and c_j are constants, then we would modify the generative process so that axioms of the form $t(c_i)$, $arg1(c_i) = c_j$, or $arg2(c_i) = c_j$ are more likely to be generated if c_i or c_j are in the list of recently-mentioned entities. And whenever an axiom of the form $t(c_i)$, $arg1(c_i) = c_j$, or $arg2(c_i) = c_j$ is generated, the context variable is modified so that the recently-mentioned entities now include c_i and c_j. This context variable is not discarded after finishing the generation of a sentence. Instead, it continues to influence the generative process of subsequent sentences. Such a model of context would enable the generation of inter-sentential anaphora: If an axiom of the form $arg1(c_i) = c_j$ or $arg2(c_i) = c_j$ is generated, where c_j is among the recently-mentioned entities in the context, then with some probability, replace c_j with a declaration of an anaphoric object, such as $\exists x(::: \wedge ref(x))$ where the existential is instantiated with c_j. Our grammar could later render $ref(x)$ as a pronoun such as "it." As this method would not fundamentally distinguish between inter- and intra-sentential anaphora, it would replace our earlier approach for parsing intra-sentential anaphora. In this approach, the logical forms for "A cat walked into the room" and "The cat purred" would look like:

$$\exists c(cat(c) \wedge U_1(c) \wedge \exists R(R = r(room(r) \wedge U_2(r)) \wedge size(R) = 1 \wedge \exists r(R(r) \wedge \exists w(arg1(w) = c \wedge walk(w) \wedge past(w) \wedge arg2(w) = r)))),$$
$$\exists C(C = c(cat(c) \wedge U_3(c)) \wedge size(C) = 1 \wedge \exists c(C(c) \wedge \exists p(arg1(p) = c \wedge purr(p) \wedge past(p)))).$$

Notice that "the room" is represented as a unique room within the universe of discourse U_2. The set R is defined as the set of all rooms within the universe of discourse given by the set U_2, and the size of R is exactly 1. The phrase "the cat" is also represented in the same way. The set C is the set of all cats within the set U_3, and the size of C is exactly 1. The universe of discourse U_3 has narrowed from U_1 so that there is exactly one cat in U_3. This model of context would be a

more realistic model for the way humans generate language naturally, as compared to PWM currently.

5

END-TO-END EXPERIMENTS

In the previous two chapters, we presented the two components of PWL: the reasoning module and language module. Here, we provide qualitative and quantitative results on experiments that evaluate the properties and capabilities of PWL end-to-end. In section 5.1, we showcase examples of sentences with syntactic ambiguities, and how PWL is able to use its knowledge acquired from previously-read sentences to resolve those ambiguities. In section 5.3, we apply PWL to the out-of-domain question-answering task in ProofWriter (Tafjord, Dalvi, and Clark, 2021) and achieve perfect zero-shot accuracy when using intuitionistic logic. However, since the sentences in ProofWriter are simple in structure, being automatically generated from templates, we create a new question-answering dataset called FictionalGeoQA, consisting of marginally more syntactically-complex (but still overall simple) sentences. The dataset is designed to be robust against algorithms that rely on simple heuristics to answer questions, and thus to more accurately measure their reasoning ability relative to other datasets. In section 5.4, we describe this dataset in further detail and show that PWL outperforms current state-of-the-art baselines.

5.1 RESOLVING SYNTACTIC AMBIGUITIES

PREPOSITIONAL PHRASE ATTACHMENT: In this section, we showcase some examples of sentences with syntactic ambiguity which are resolved via the consideration of background knowledge. Each example serves to illustrate PWL's reading process step-by-step, and also demonstrates how PWL is able to capitalize on knowledge acquired from previously-read sentences to resolve syntactic ambiguities when reading new sentences. However, we also showcase some examples that highlight shortcomings of PWL that arise from, for example, the simplicity of the prior on the theory and proofs.

Recall that PWL reads sentences in two steps. In the first step, given a new (unseen) sentence y, find the k-best values of the logical form x, according to the likelihood $p(y \mid x, \Sigma)$. In the second step, for each of the k logical forms, compute its prior $p(x \mid T)$ and rerank them according to the posterior:

$$p(x \mid \Sigma, y, y, T) \propto p(x \mid T)p(y \mid x, \Sigma).$$

Figure 19: An example where PWL reads the sentence "Sally caught a butterfly with a net," which is a classical example of a sentence with prepositional phrase attachment ambiguity: "with a net" could either attach to "butterfly" or "caught." In PWL, "reading" a sentence is divided into two stages: (1) find the 4 most likely logical forms, ignoring the prior probability of each logical form conditioned on the theory, and (2) for each logical form in the list, computing its prior probability conditioned on the theory and then reranking the list accordingly. The output of the first stage is shown in the top table, and the output of the second stage is shown in the bottom table. In this example, PWL has previously read "No butterfly has a net," and added its logical form to the theory. As a result, the reasoning module is unable to find a theory that explains the logical form where the butterfly has the net, and so the prior probability of that logical form is zero. The log probabilities in the bottom table are unnormalized.

The reason for this two-stage process is the following: In the upper bound for the parser (as discussed in section 4.3.2, in equations 87, 88, and 89), we need to define $h^n_x(X)$, which is the upper bound on the semantic prior $\log p(x^n)$ *at node* n of the derivation tree. It is easier to define this bound when the semantic prior is simple, so that it satisfies the property that for any logical form x, and for any derivation tree node n with child c, $p(x^n) \leq p(x^c)$. This property *is* satisfied for the prior that we use in the semantic parsing experiments in section 4.4, but it is *not* satisfied by the prior on logical forms defined by the reasoning module (discussed in chapter 3). For example, the addition of a negation can dramatically increase or decrease the prior probability of a logical form. As a result, in the semantic parsing experiments in section 4.4, the branch-and-bound algorithm is able to directly find the k-best logical forms that maximize the full posterior $p(x \mid x, y, y, T)$. However, for the other experiments of this thesis, PWL uses a trivial upper bound on the semantic prior $h^n_x(X) = 0 > \log p(x^n)$, and so the branch-and-bound algorithm finds the k-best logical forms that maximize the likelihood $p(y \mid x, x, y)$ (without consideration of the semantic prior). Then in the second step, PWL computes the prior of each of the k logical forms $p(x \mid T)$, and reranks them according to their full posterior.

In this first example, we demonstrate how PWL can utilize previously-acquired knowledge to resolve the prepositional phrase attachment ambiguity in the sentence "Sally caught a butterfly with a net." The prepositional phrase "with a net" can either attach to the noun "butterfly" or to the verb "caught." If it attaches to "butterfly," the semantic interpretation is that the butterfly has a net. If it attaches to "caught," the semantic interpretation is that Sally used a net to catch a butterfly. We first consider PWL reading the sentence without any other sentences or axioms, aside from those of the seed training set. In the first stage of reading, the branch-and-bound algorithm visits 7387 states and finds the top 4 logical forms that maximize the *likelihood*:

A. `x6( x5(name(x5)  arg1_of(x6)=x5  arg2(x5)="Sally")`
`x10(net(x10))  x9(butterfly(x9))  x1(has(x1)`
`arg2(x1)=x10  arg1_of(x9)=x1)  x1(arg1(x1)=x6`
`catch(x1)  past(x1)  arg2(x1)=x9)))),`

with log likelihood `-31.834222`. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_9) that has a net (x_{10}), and Sally caught that butterfly.

B. `x6( x5(name(x5)  arg1_of(x6)=x5  arg2(x5)="Sally")`
`x10(butterfly(x10))  x9(net(x9))  x1(arg1(x1)=x6`
`catch(x1)  past(x1)  arg2(x1)=x10`
`x8(use_instrument(x8)  arg2(x8)=x9`
`arg1_of(x1)=x8))))),`

with log likelihood -34.767277. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_{10}), and Sally used the net (x_9) to catch the butterfly.

C. $x_6($ x_5(name(x_5) arg1_of(x_6)=x_5 arg2(x_5)="Sally")
x_{10}(net(x_{10}) x_9(butterfly(x_9) x_1(has(x_1)
arg2(x_1)=x_{10} arg1_of(x_9)=x_1) x_1(arg2(x_1)=x_6
catch(x_1) past(x_1) arg1(x_1)=x_9))))),

with log likelihood -35.544891. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_9) that has a net (x_{10}), and *the butterfly caught Sally*.

D. $x_6($ x_5(name(x_5) arg1_of(x_6)=x_5 arg2(x_5)="Sally")
x_{10}(butterfly(x_{10}) x_9(net(x_9) x_1(arg2(x_1)=x_6
catch(x_1) past(x_1) arg1(x_1)=x_{10}
x_8(use_instrument(x_8) arg2(x_8)=x_9
arg1_of(x_1)=x_8))))).

with log likelihood -38.477945. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_{10}), and *the butterfly used the net (x_9) to catch Sally*.

This sentence is a prototypical example of prepositional phrase attachment ambiguity, where the prepositional phrase "with a net" may attach to either the noun "butterfly" or the verb "caught." Without any information from the semantic prior, the parser prefers the closer attachment (i.e. the butterfly has the net).

In the second stage of the reading process, the semantic prior is computed for each of the above four logical forms, using the sampling method described in section 3.3.1, with 400 iterations of MH. The above list is then reranked according to the sum of the log semantic prior and the log likelihood:

A. log likelihood -31.834222 + log prior -2212.409332
= log posterior -2244.243554,

C. log likelihood -35.544891 + log prior -2209.566363
= log posterior -2245.111253,

B. log likelihood -34.767277 + log prior -2212.409332
= log posterior -2247.176609,

D. log likelihood -38.477945 + log prior -2212.591654
= log posterior -2251.069599.

The most probable logical form in the reranked list did not change. Note the computed semantic prior is unnormalized, since we only aim to compare the relative probabilities of the logical forms in the list, and a constant normalization term has no effect on the ranking.

Next, consider first reading the sentence "No butterfly has a net." The parser returns the logical form

```
x4(butterfly(x4)    x5(net(x5)
    x1(arg1(x1)=x4   has(x1)   present(x1)   arg2(x1)=x5))),
```

which has the meaning that there does not exist a butterfly (x_4) that has a net (x_5). We add this logical form to the theory, and again attempt to read "Sally caught a butterfly with a net."

The parser of PWL preserves information encoded in the aspect and tense of verbs. They are parsed into terms such as `past(X)` which indicates that the event `X` occurred in the past. However, in order to correctly handle such terms in the reasoning module, a model of time is required. So in order to avoid overly complicating the experiments in this chapter, PWL discards terms that indicate the aspect and tense, before providing the logical forms to the reasoning module. If we did not discard these terms, PWL would naively interpret "No butterfly has a net" to be true in the present but not necessarily in the past, which is when Sally caught the butterfly. There are also many interesting and difficult questions regarding the correct handling of aspect and tense, since the reference point which indicates the "present time" can change with context: The use of the past tense in one sentence may have a meaning distinct from the use of the past tense in a later sentence. This reference point is referred to as *origo* in pragmatics.

In reading "Sally caught a butterfly with a net," the first stage (i.e. the branch-and-bound algorithm) returns the same list of 4 logical forms as above, it only maximizes the likelihood, without consideration of the semantic prior. However, in the second stage of reading, the semantic prior is computed for each of the logical forms, and reranked accordingly. The resulting ranked list is now:

B. `log likelihood -34.767277 + log prior -2294.538125 = log posterior -2329.305402,`

D. `log likelihood -38.477945 + log prior -2294.538125 = log posterior -2333.016070,`

C. `log likelihood -35.544891 + log prior -inf = log posterior -inf,`

A. `log likelihood -31.834222 + log prior -inf = log posterior -inf.`

The interpretations where the butterfly has the net is now impossible (the proof initialization algorithm is unable to find a valid proof of the logical form that is consistent with the theory). The most probable logical form in the reranked list is the correct interpretation where Sally used the net to capture the butterfly. A summary of the above example is shown in figure 19.

In the above example, the sentence "No butterfly has a net" provided a hard constraint on the possible interpretations of "Sally caught a butterfly with a net." Instead of reading "No butterfly has a net," suppose PWL first reads "Sally uses a net," which provides a softer constraint. The semantic parse of this sentence is:

```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Sally")
     x7(net(x7)    x1(arg1(x1)=x6   use(x1)   present(x1)
    arg2(x1)=x7))),
```

which has the meaning that there exists something named "Sally" (x_6) and there exists a net (x_7) such that Sally uses the net. We add this logical form to the theory (instead of the logical form for "No butterfly has a net"), and then again attempt to read "Sally caught a butterfly with a net." The first stage of reading returns the same list of logical forms as above, since the branch-and-bound algorithm is only maximizing the likelihood. The second state of reading, however, returns the following reranked list:

A. log likelihood -31.834222 + log prior -2440.877451
   = log posterior -2472.711673,

C. log likelihood -35.544891 + log prior -2440.232007
   = log posterior -2475.776897,

B. log likelihood -34.767277 + log prior -2444.451668
   = log posterior -2479.218944,

D. log likelihood -38.477945 + log prior -2444.451668
   = log posterior -2482.929613.

Notice that this reranking step did not significantly change the relative probabilities of the candidate logical forms, and in fact, the incorrect interpretation is still highest-ranked. The reason for this is that the instrumentative sense of the preposition "with" is interpreted as a `use_instrument` event, whereas the verb "use" is interpreted as a `use` event. In order to correctly identify the equivalence of these events, PWL would need an axiom such as:

```
∀x∀y(∃f(arg1(f)=x
    ^ ∃u(use_instrument(u) ^ arg1(u)=f ^ arg2(u)=y))
    ! ∃u(use(u) ^ arg1(u)=x ^ arg2(u)=y)).
```

That is, PWL would need an axiom that if x uses an instrument y in some event or action f, then x uses y. Since such an axiom is not present in the seed axioms of the theory, PWL fails to correctly disambiguate the prepositional phrase ambiguity.

Now we consider a slightly different example: Suppose that instead of reading "No butterfly has a net" or "Sally uses a net," PWL first

reads "A butterfly has a spot." This sentence would avoid the problem mentioned above. The semantic parse of this sentence is:

```
x4(butterfly(x4)    x5(spot(x5)
    x1(arg1(x1)=x4   has(x1)   present(x1)   arg2(x1)=x5))),
```

which has the meaning that there exists a butterfly (x_4) that has a spot (x_5). We add this logical form to the theory (instead of the logical form for "No butterfly has a net" or "Sally uses a net"), and then attempt to read "Sally caught a butterfly with a spot." The first stage of reading returns the following list of logical forms that maximize the likelihood:

A.
```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Sally")
     x10(spot(x10)    x9(butterfly(x9)    x1(has(x1)
    arg2(x1)=x10   arg1_of(x9)=x1)    x1(arg1(x1)=x6
  catch(x1)   past(x1)   arg2(x1)=x9)))),
```

with log likelihood `-31.834222`. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_9) that has a spot (x_{10}), and Sally caught that butterfly.

B.
```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Sally")
     x10(butterfly(x10)    x9(spot(x9)
   x1(arg1(x1)=x6   catch(x1)   past(x1)   arg2(x1)=x10
     x8(use_instrument(x8)   arg2(x8)=x9
  arg1_of(x1)=x8))))),
```

with log likelihood `-34.767277`. This logical form has the meaning: there exists something named "Sally" (x_6), a butterfly (x_{10}), and a spot (x_9), and Sally used the spot to catch the butterfly.

C.
```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Sally")
     x10(spot(x10)    x9(butterfly(x9)    x1(has(x1)
    arg2(x1)=x10   arg1_of(x9)=x1)    x1(arg2(x1)=x6
  catch(x1)   past(x1)   arg1(x1)=x9)))),
```

with log likelihood `-35.544891`. This logical form has the meaning: there exists something named "Sally" (x_6), and there exists a butterfly (x_9) that has a spot (x_{10}), and *the butterfly caught Sally*.

D.
```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Sally")
     x10(butterfly(x10)    x9(spot(x9)
   x1(arg2(x1)=x6   catch(x1)   past(x1)   arg1(x1)=x10
     x8(use_instrument(x8)   arg2(x8)=x9
  arg1_of(x1)=x8))))).
```

with log likelihood `-38.477945`. This logical form has the meaning: there exists something named "Sally" (x_6), a butterfly (x_{10}), and a spot (x_9), and *the butterfly used the spot to catch Sally*.

The second stage of reading computes the semantic prior for each of the above logical forms and reranks them:

A. log likelihood -31.834222 + log prior -2329.093733
= log posterior -2360.927955,

C. log likelihood -35.544891 + log prior -2329.786880
= log posterior -2365.331771,

B. log likelihood -34.767277 + log prior -2378.812990
= log posterior -2413.580266,

D. log likelihood -38.477945 + log prior -2379.013660
= log posterior -2417.491606.

In the reranked list, both interpretations where the prepositional phrase attaches to the noun (i.e. the butterfly has the spot) are now much more probable than both interpretations where the prepositional phrase attaches to the verb (i.e. Sally uses the spot to catch the butterfly). But unlike the earlier example with the net, it is still possible that a spot can be used as an instrument to catch something, if we know nothing else about spots. As such, the reasoning module is able to construct theories that describe these possibilities, however unlikely.

PRONOMINAL RESOLUTION: We also showcase an example where PWL resolves ambiguity in pronominal resolution in the sentence "A butterfly has a spot and it is blue," where "it" can refer to either the butterfly or the spot. We will show that PWL is able to acquire knowledge from other sentences, and utilize that knowledge to resolve the ambiguous pronoun "it." First consider parsing the sentence "A butterfly has a spot and it is blue" without any axioms or other sentences aside from the seed training set. The branch-and-bound algorithm finds the 4-best logical forms that maximize the log likelihood, after visiting 27,269 states:

A. x_4(butterfly(x_4) x_5(spot(x_5) x_1(arg1(x_1)=x_4
has(x_1) present(x_1) arg2(x_1)=x_5))) x_4(ref(x_4)
x_1(arg1(x_1)=x_4 blue(x_1) present(x_1))),

with log likelihood -40.408566. This logical form has the meaning: there exists a butterfly that has a spot, and there exists an object with special anaphoric type ref (representing "it") that is blue.

B. x_4(butterfly(x_4) x_5(spot(x_5) x_1(arg1(x_1)=x_4
has(x_1) present(x_1) arg2(x_1)=x_5))) x_4(ref(x_4)
x_1(arg2(x_1)=x_4 blue(x_1) present(x_1))),

with log likelihood -41.982132. This logical form has the meaning: there exists a butterfly that has a spot, an object x_4 with special anaphoric type ref (representing "it"), and there exists an event with type blue whose second argument is x_4. This logical form is spurious since blue is meant to be a property, and its arity should be 1.

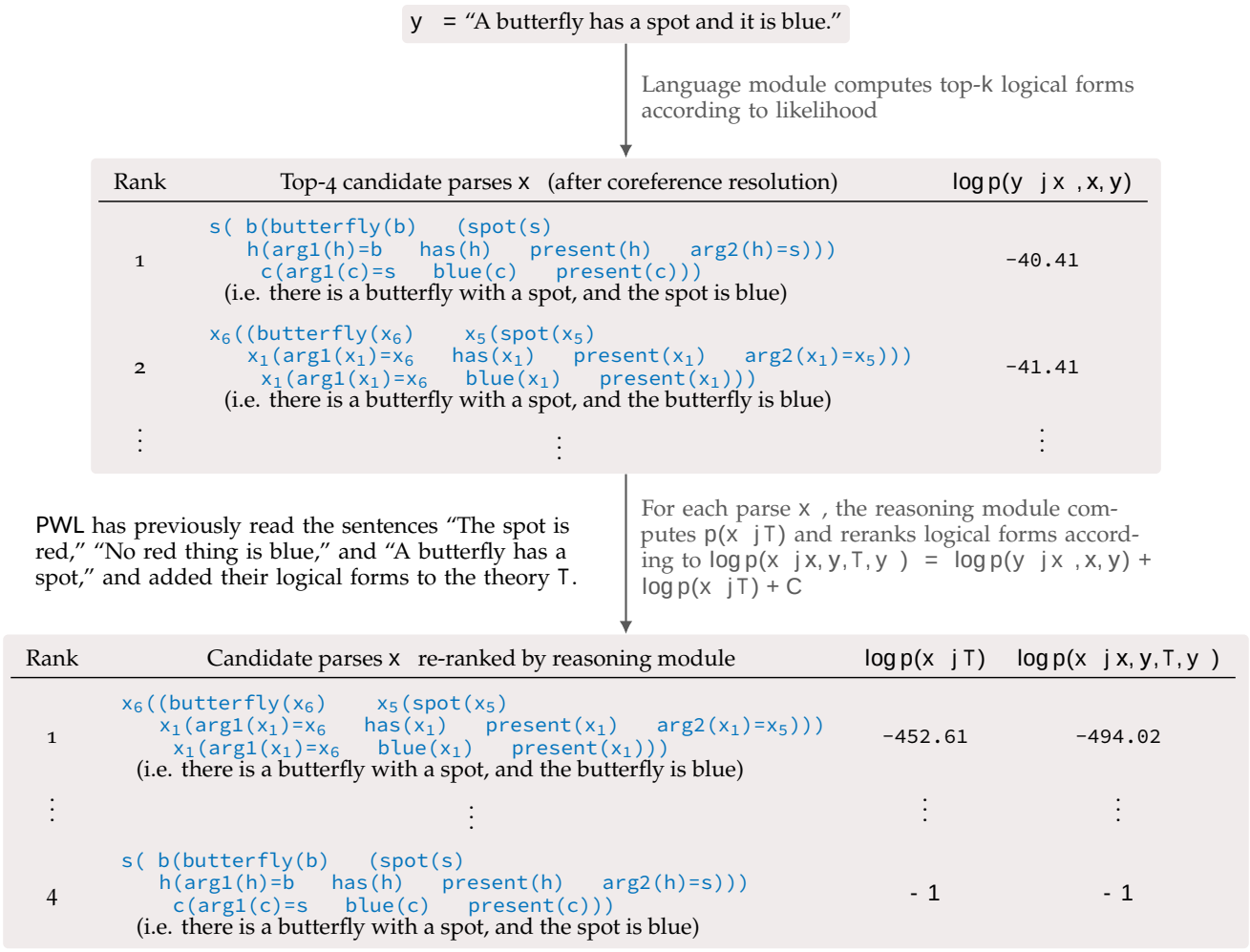


Figure 20: An example where PWL reads the sentence "A butterfly has a spot and it is blue," which is an example of a sentence with an ambiguous pronoun: "it" could either refer to "butterfly" or "spot." The output of the first stage of reading is shown in the top table (after intra-sentential coreference resolution), and the output of the second stage is shown in the bottom table. In this example, PWL has previously read "The spot is red," "No red thing is blue," and "A butterfly has a spot," and added their logical form to the theory. As a result, the reasoning module is unable to find a theory where the spot is blue, and so the prior probability of that logical form is zero. The log probabilities in the bottom table are unnormalized.

C. x_4(butterfly(x_4) x_5(spot(x_5) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_5))) x_4(ref(x_4) x_1(arg1(x_1)=x_4 mass(x_1) present(x_1))),

with log likelihood -42.157300. This logical form has the meaning: there exists a butterfly that has a spot, an object x_4 with special anaphoric type ref (representing "it"), and there exists an event with type mass whose first argument is x_4.

D. x_4(butterfly(x_4) x_5(spot(x_5) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_5))) x_4(ref(x_4) x_1(arg1(x_1)=x_4 terrestrial(x_1) present(x_1))).

with log likelihood -43.325155. This logical form has the meaning: there exists a butterfly that has a spot, an object x_4 with special anaphoric type ref (representing "it"), and there exists an event with type terrestrial whose first argument is x_4.

The parser then performs intra-sentential coreference resolution described in section 4.5.1, and the resulting top-4 logical forms are:

A.1. x_6(x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg1(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: there exists a butterfly that has a spot, and the spot is blue.

A.2. x_6((butterfly(x_6) x_5(spot(x_5) x_1(arg1(x_1)=x_6 has(x_1) present(x_1) arg2(x_1)=x_5))) x_1(arg1(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: there exists a butterfly that has a spot, and the butterfly is blue.

B. x_6(x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg2(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: there exists a butterfly that has a spot, and there exists an event with type blue whose second argument is the spot (this is the same spurious logical form as above).

C. x_6(x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg1(x_1)=x_6 mass(x_1) present(x_1))),

which has the meaning: there exists a butterfly that has a spot, and there exists an event with type mass whose first argument is the spot.

The log likelihoods above are fairly close, as a result of the fact that the word "blue" does not appear within a sentence of the seed training set. Rather, it is specified as a standalone adjective. Thus, the parser is

generally less certain about its use within a sentence. In addition, the grammar does not currently incorporate the fact that when adjectives are used predicatively (i.e. as a predicate), the corresponding predicate in the logical form should be unary. The production rules need to be modified in order to incorporate this observation. At the second stage of reading, PWL computes the semantic prior for each of the above logical forms and reranks them:

A.1 log likelihood -40.408566 + log prior -122.844591
= log posterior -163.253157,

A.2 log likelihood -41.408566 + log prior -122.844591
= log posterior -164.253157,

B. log likelihood -41.982132 + log prior -122.844591
= log posterior -164.826723,

C. log likelihood -42.157300 + log prior -122.844591
= log posterior -165.001892.

Now consider the case where PWL first reads the sentences "The spot is red," and "No red thing is blue," before reading the sentence "A butterfly has a spot and it is blue." PWL parses "The spot is red" into the logical form:

$$x_6(x_6= x_3\text{spot}(x_3) \quad \text{size}(x_6)=1 \quad x_5(x_6(x_5) \\ x_1(\text{arg1}(x_1)=x_5 \quad \text{red}(x_1) \quad \text{present}(x_1)))),$$

which has the meaning: there exists a set x_6 which is the set of all spots, x_6 has size 1, x_5 is an element of x_6, and x_5 is red. Similarly, PWL parses "No red thing is blue" into the logical form:

$$x_4(x_1(\text{red}(x_1) \quad \text{arg1\_of}(x_4)=x_1) \quad \text{object}(x_4) \\ x_1(\text{arg1}(x_1)=x_4 \quad \text{blue}(x_1) \quad \text{present}(x_1))),$$

which has the meaning: there does not exist an object that is both red and blue. Note that PWL assumes that all objects have type `object`. Both of the above logical forms are added to the theory, and we again try to read the sentence "A butterfly has a spot and it is blue." The first stage of parsing is unchanged from before, but the second stage of parsing produces the following ranked list of logical forms:

A.2 log likelihood -41.408566 + log prior -452.610736
= log posterior -494.019302,

C. log likelihood -42.157300 + log prior -452.644638
= log posterior -494.801938,

B. log likelihood -41.982132 + log prior -452.876440
= log posterior -494.858571,

A.1 `log likelihood -40.408566 + log prior -inf = log posterior -inf.`

PWL correctly infers that the interpretation where the spot is blue is impossible, and the most probable logical form is the one in which "it" refers to the butterfly rather than the spot. Figure 20 summarizes this example.

Here, we showcase an example where PWL is limited by the simplicity of the prior on the theory and proofs. Consider first reading the sentences "The spot is red," "No red thing is blue," and then "If a butterfly has a spot, then it is blue." The first stage of parsing returns the following list of most likely logical forms:

A. `( x4(butterfly(x4)  x5(spot(x5)  x1(arg1(x1)=x4 has(x1)  present(x1)  arg2(x1)=x5)))  x4(ref(x4) x1(arg1(x1)=x4  blue(x1)  present(x1)))),`

with log likelihood `-47.038724`. This logical form has the meaning: If there exists a butterfly (x_4) and a spot (x_5), and the butterfly has the spot, then there exists an object of anaphoric type `ref` (x_4) that is blue.

B. `( x4(butterfly(x4)  x5(spot(x5)  x1(arg1(x1)=x4 has(x1)  present(x1)  arg2(x1)=x5)))  x4(ref(x4) x1(arg2(x1)=x4  blue(x1)  present(x1)))),`

with log likelihood `-48.612290`. This logical form has the meaning: If there exists a butterfly (x_4) and a spot (x_5), and the butterfly has the spot, then there exists an object of anaphoric type `ref` (x_4), and an event of type `blue` such that its second argument is x_4. As with the previous example, this logical form is spurious since the `blue` event is unary.

C. `( x4(butterfly(x4)  x5(spot(x5)  x1(arg1(x1)=x4 has(x1)  present(x1)  arg2(x1)=x5)))  x4(ref(x4) x1(arg1(x1)=x4  mass(x1)  present(x1)))),`

with log likelihood `-48.787458`. This logical form has the meaning: If there exists a butterfly (x_4) and a spot (x_5), and the butterfly has the spot, then there exists an object of anaphoric type `ref` (x_4), and an event of type `mass` such that its first argument is x_4.

D. `( x4(butterfly(x4)  x5(spot(x5)  x1(arg1(x1)=x4 has(x1)  present(x1)  arg2(x1)=x5)))  x4(ref(x4) x1(arg1(x1)=x4  terrestrial(x1)  present(x1)))).`

with log likelihood `-49.955313`. This logical form has the meaning: If there exists a butterfly (x_4) and a spot (x_5), and the butterfly has the spot, then there exists an object of anaphoric type `ref` (x_4), and an event of type `terrestrial` such that its first argument is x_4.

The output of intra-sentential coreference resolution is:

A.1. $x_6($ x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg1(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: for all spots, if there exists a butterfly that has that spot, the spot is blue.

A.2. x_6(butterfly(x_6) x_5(spot(x_5) x_1(arg1(x_1)=x_6 has(x_1) present(x_1) arg2(x_1)=x_5)) x_1(arg1(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: for all butterflies, if the butterfly has a spot, the butterfly is blue.

B. $x_6($ x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg2(x_1)=x_6 blue(x_1) present(x_1))),

which has the meaning: for all spots, if there exists a butterfly that has that spot, there exists an event with type blue whose second argument is the spot (this is the same spurious logical form as above).

C. $x_6($ x_4(butterfly(x_4) (spot(x_6) x_1(arg1(x_1)=x_4 has(x_1) present(x_1) arg2(x_1)=x_6))) x_1(arg1(x_1)=x_6 mass(x_1) present(x_1))),

which has the meaning: for all spots, if there exists a butterfly that has that spot, there exists an event with type mass whose first argument is the spot.

The second stage of reading then reranks the above list according to the semantic prior of each logical form:

A.1 log likelihood -47.038724 + log prior -451.290984 = log posterior -498.329707,

A.2 log likelihood -48.038724 + log prior -451.316548 = log posterior -499.355272,

B. log likelihood -48.612290 + log prior -451.281382 = log posterior -499.893672,

C. log likelihood -48.787458 + log prior -452.038785 = log posterior -500.826244.

Unlike the last example, the interpretation where the spot is blue is no longer impossible. In fact, it is now the most probable interpretation. The reason for this is that PWL is still able to construct consistent theories in which the first logical form in the above list is an axiom: the axiom would be vacuously true if there do not exist any butterflies with spots.

However, if we additionally provide the background sentence "A butterfly has a spot" (in addition to "The spot is red" and "No red thing is blue"), and then re-run the above reading procedure for "If a butterfly has a spot, then it is blue," the output of the second stage would become:

A.2 `log likelihood -48.038724 + log prior -637.068882 = log posterior -685.107606,`

B. `log likelihood -48.612290 + log prior -637.017189 = log posterior -685.629478,`

C. `log likelihood -48.787458 + log prior -638.144783 = log posterior -686.932241,`

A.1 `log likelihood -47.038724 + log prior -inf = log posterior -inf.`

Now, the interpretation where the spot is blue is correctly given zero posterior probability, and the most probable logical form is the one where the butterfly is blue. However, this example highlights a difference in the prior probability of logical forms that are vacuously true. Humans are very unlikely to generate such logical forms, but they are not discouraged by PWM.

LEXICAL AMBIGUITY: Here, we showcase an example where PWL uses semantics to resolve lexical ambiguity. In this example, in the sentence "Minas Tirith is the largest city," the word "largest" may either refer to maximizing area or population. But if PWL is first given sentences that specify the area of Minas Tirith and another city, Pelargir, we demonstrate that PWL is able to correctly synthesize this information, along with axioms that define maximality, in order to resolve the lexical ambiguity.

In order to correctly disambiguate the meaning of "largest" in this example, the reasoning module first needs to know the meaning of maximality in order to reason about it. In all of the experiments described in this chapter, the following two axioms are added to the theory before reading any sentences. They define the meaning of maximality and minimality (represented as events of type `greatest` and `least`):

```
x1 x2 x3(
 x4(greatest(x2)(x4)   arg1(x4)=x3   arg2(x4)=x1)
   x3(x1)    x4(x2(x1)(x4)    x5(x3(x5)    x6(x2(x5)(x6)
    (number(x4)   ge(x4,x6))    x7(measure(x4)   arg1(x4)=x7
      x8(measure(x6)    x9(arg2(x4)=x9   arg2(x6)=x9)
      arg1(x6)=x8   ge(x7,x8))))))),
```

which states that for any object x_1 that maximizes the function x_2 over the set x_3 (i.e. there is an event of type `greatest(x2)`, where the first

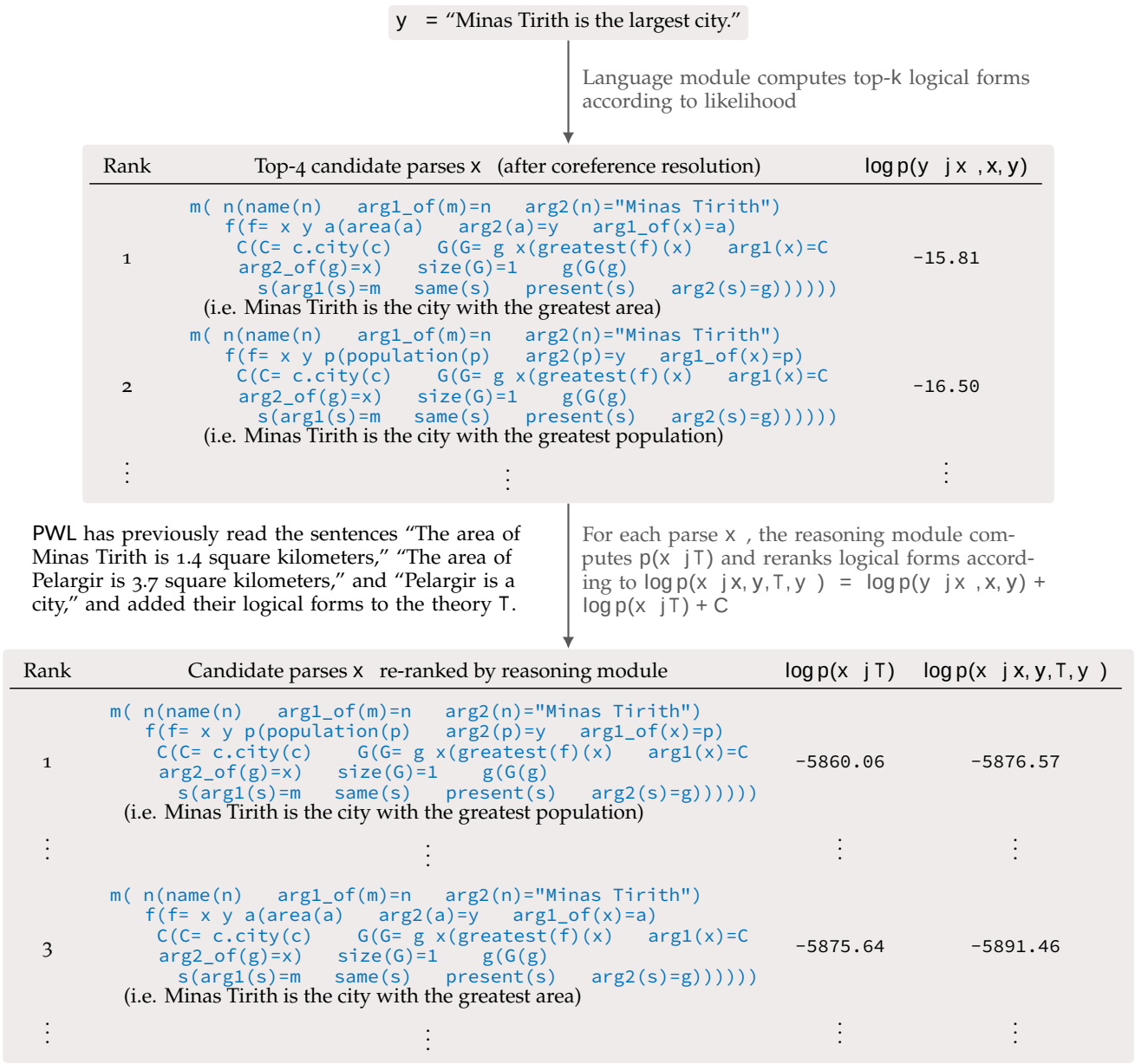


Figure 21: An example where PWL reads the sentence "Minas Tirith is the largest city," which is an example of a sentence with an ambiguous word: "largest city" could either refer to the city with the largest area or the largest population. The output of the first stage of reading is shown in the top table, and the output of the second stage is shown in the bottom table. In this example, PWL has previously read "The area of Minas Tirith is 1.4 square kilometers," "The area of Pelargir is 3.7 square kilometers," and "Pelargir is a city," and added their logical form to the theory. As a result, the reasoning module only finds lower probability theories in which Minas Tirith is the city with the largest area (an example of such a theory is one where there are two cities named Minas Tirith, one of which has area at least 3.7 sq km). The log probabilities in the bottom table are unnormalized.

argument is x_1, the second argument is x_3), then x_1 is an element of x_3, and for any element x_5 of x_3, the value of the function x_2 of x_1 is greater than or equal to the value of the function x_2 of x_5 (where the predicate `ge` denotes greater than or equal). This axiom also handles cases where the two quantities being compared are quantities with units (i.e. instances of `measure` whose first argument is the numerical quantity and second argument is the unit). In which case, the quantities are only compared if they have the same unit x_9. Similarly, the second axiom defines the meaning of minimality:

```
x1 x2 x3(
 x4(least(x2)(x4)   arg1(x4)=x3   arg2(x4)=x1)
  x3(x1)   x4(x2(x1)(x4)   x5(x3(x5)   x6(x2(x5)(x6)
   (number(x4)   ge(x6,x4))   x7(measure(x4)   arg1(x4)=x7
    x8(measure(x6)   x9(arg2(x4)=x9   arg2(x6)=x9)
    arg1(x6)=x8   ge(x8,x7)))))))).
```

In principle, additional seed axioms could easily be added to encode further background knowledge.

Now, consider the example where PWL parses the sentence "Minas Tirith is the largest city," without having read any other sentences. The branch-and-bound algorithm returns the 4 logical forms that maximize the likelihood, after visiting 3380 states:

A. `x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Minas Tirith")`
`x26(x26= x8 x9 x27(area(x27)   arg2(x27)=x9   arg1_of(x8)=x27)`
`x25(x25= x12city(x12))   x24(x24= x3 x5(greatest(x26)(x5)`
`arg1(x5)=x25   arg2_of(x3)=x5)   size(x24)=1   x23(x24(x23)`
`x1(arg1(x1)=x6   same(x1)   present(x1)   arg2(x1)=x23))))))`,

with log likelihood `-15.812698`. This logical form has the meaning: Minas Tirith is the city with the greatest area.

B. `x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Minas`
`Tirith")   x26(x26= x8 x9 x27(population(x27)`
`arg2(x27)=x9   arg1_of(x8)=x27)   x25(x25= x12city(x12)`
`x24(x24= x3 x5(greatest(x26)(x5)   arg1(x5)=x25`
`arg2_of(x3)=x5)   size(x24)=1   x23(x24(x23)   x1(arg1(x1)=x6`
`same(x1)   present(x1)   arg2(x1)=x23))))))`,

with log likelihood `-16.505754`. This logical form has the meaning: Minas Tirith is the city with the greatest population.

C. `x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Minas Tirith")`
`x26(x26= x8 x9 x27(area(x27)   arg2(x27)=x9   arg1_of(x8)=x27)`
`x25(x25= x12city(x12))   x24(x24= x3 x5(greatest(x26)(x5)`
`arg1(x5)=x25   arg2_of(x3)=x5)   size(x24)=1   x23(x24(x23)`
`x1(arg2(x1)=x6   same(x1)   present(x1)   arg1(x1)=x23))))))`,

with log likelihood `-17.198972`. This logical form has the same meaning as the first logical form in this list, but the arguments of

the same event are swapped (the parser is not aware that same is symmetric).

D. $x_6($ x_5(name(x_5) arg1_of(x_6)=x_5 arg2(x_5)="Minas Tirith") x_{26}(x_{26}= x_8 x_9 x_{27}(population(x_{27}) arg2(x_{27})=x_9 arg1_of(x_8)=x_{27}) x_{25}(x_{25}= x_{12}city(x_{12}) x_{24}(x_{24}= x_3 x_5(greatest(x_{26})(x_5) arg1(x_5)=x_{25} arg2_of(x_3)=x_5) size(x_{24})=1 x_{23}(x_{24}(x_{23}) x_1(arg2(x_1)=x_6 same(x_1) present(x_1) arg1(x_1)=x_{23}))))))).

with log likelihood -17.892028. This logical form has the same meaning as the second logical form in this list, but the arguments of the same event are swapped (the parser is not aware that same is symmetric).

In the second stage of reading, PWL computes the semantic prior for each of the above logical forms and reranks them:

A. log likelihood -15.812698 + log prior -2378.425525 = log posterior -2394.238222,

B. log likelihood -16.505754 + log prior -2378.157261 = log posterior -2394.663015,

C. log likelihood -17.198972 + log prior -2377.854980 = log posterior -2395.053952,

D. log likelihood -17.892028 + log prior -2378.592579 = log posterior -2396.484607.

Unsurprisingly, since the theory has no axioms from prior observations, the ranking does not change. The most probable interpretation of "largest" is that of maximizing area.

Next, suppose PWL first reads the sentences "The area of Minas Tirith is 1.4 square kilometers," "The area of Pelargir is 3.7 square kilometers," and "Pelargir is a city." PWL parses "The area of Minas Tirith is 1.4 square kilometers" into the logical form:

x_{16}(x_{10}(name(x_{10}) arg1_of(x_{16})=x_{10} arg2(x_{10})="Minas Tirith") x_{15}(x_{15}= x_3 x_5(area(x_5) arg1(x_5)=x_{16} arg2_of(x_3)=x_5) size(x_{15})=1 x_{14}(x_{15}(x_{14}) x_{18}(kilometer(x_{18}) x_{17}(measure(x_{17}) arg1(x_{17})=1.4 arg2(x_{17})=x_{18} x_1(arg1(x_1)=x_{14} same(x_1) present(x_1) arg2(x_1)=x_{17}))))))),

which has the meaning: There exists an object named "Minas Tirith" x_{16}, which has a unique area x_{14}. x_{14} is identical to x_{17} which is an instance of measure, whose quantity is 1.4 and unit is x_{18} which

has type kilometer. PWL parses "The area of Pelargir is 3.7 square kilometers" into the logical form:

```
x16( x10(name(x10)   arg1_of(x16)=x10   arg2(x10)="Pelargir")
     x15(x15= x3 x5(area(x5)   arg1(x5)=x16   arg2_of(x3)=x5)
     size(x15)=1   x14(x15(x14)   x18(kilometer(x18)
     x17(measure(x17)   arg1(x17)=3.7   arg2(x17)=x18
     x1(arg1(x1)=x14   same(x1)   present(x1)   arg2(x1)=x17))))),
```

which has the meaning: There exists an object named "Pelargir" x_{16}, which has a unique area x_{14}. x_{14} is identical to x_{17} which is an instance of measure, whose quantity is 3.7 and unit is x_{18} which has type kilometer. PWL parses "Pelargir is a city" into the logical form:

```
x6( x5(name(x5)   arg1_of(x6)=x5   arg2(x5)="Pelargir")   x7(city(x7)
    x1(arg1(x1)=x6   same(x1)   present(x1)   arg2(x1)=x7))),
```

which has meaning: There exists an object named "Pelargir" x_6 which is equivalent to x_7 which is an instance of city. We add these logical forms to the theory and then attempt to read the sentence "Minas Tirith is the largest city" once more. The first stage of parsing is unchanged, since it is independent of the theory. But in the second stage, PWL reranks the logical forms according to their semantic prior. The resulting ranked list is:

B. log likelihood -16.505754 + log prior -5860.061336 = log posterior -5876.567091,

D. log likelihood -17.892028 + log prior -5859.448232 = log posterior -5877.340260,

A. log likelihood -15.812698 + log prior -5875.643694 = log posterior -5891.456391,

C. log likelihood -17.198972 + log prior -7922.388543 = log posterior -7939.587514.

Now, the most probable interpretation of "largest" is that which maximizes population, rather than area. PWL was (successfully) unable to construct a high-probability theory where Minas Tirith is the city that maximizes area, since there exists another city (Pelargir) with greater area, and the definition "maximality" is given by the axioms mentioned earlier. Instead, the reasoning module finds lower probability theories, such as one where there are two cities named "Minas Tirith," one of which has area 1.4 sq km, and the other has area at least 3.7 sq km. Thus, the most probable theories are those in which "largest" refers to population.

We will show in section 5.4 that this ability to exploit semantic information to resolve lexical ambiguity is helpful in question-answering, where the questions can exhibit lexical ambiguity.

5.2 REASONING OVER SIZES OF SETS

In this section, we showcase an example where PWL reads sentences that state the number of various objects, which are interpreted as sets, and inspect the sizes of those sets in the posterior samples of the theory. The example here is a simple counting problem that young children are able to solve. This example will demonstrate that PWL is capable of parsing and understanding the semantics of sentences that convey information about the number of objects of various types, and is capable of reasoning about that information. Additionally, many modern natural language understanding methods notoriously struggle with discrete reasoning such as counting. The use of a symbolic formal language and symbolic reasoning helps PWL in situations that require discrete reasoning, and generalization to larger sets is guaranteed by design.

In the first example, PWL reads the sentences "There are 30 red or blue things," and "Every fish is red or blue." Their most probable logical forms are added to the theory and then we perform 120,000 iterations of MH. At each MH sample, we record the size of the known set `x.fish(x)` (i.e. the number of fish). Figure 22 shows a histogram of the samples of this quantity. As expected, the number of fish takes any integer value between 0 and 30.

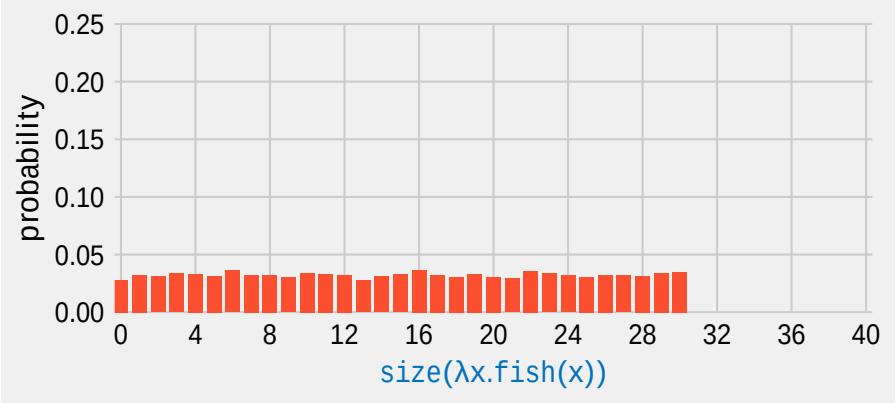


Figure 22: Histogram of the size of the set of fish (i.e. the number of fish), from the MH samples of the theory, after reading the sentences "There are 30 red or blue things," and "Every fish is red or blue."

Next, PWL reads another sentence "There are six red fish," and adds the most probable logical form to the theory. Again, we perform 120,000 iterations of MH and record the number of fish in each sample theory. These samples are shown in the histogram in figure 23. As expected, since there are six red fish, the lower bound on the number of fish is 6, as well.

Next, PWL reads the sentence "There are 24 blue fish," and adds the most probable logical form to the theory. Again, we perform 120,000 iterations of MH and record the samples of the number of fish. The

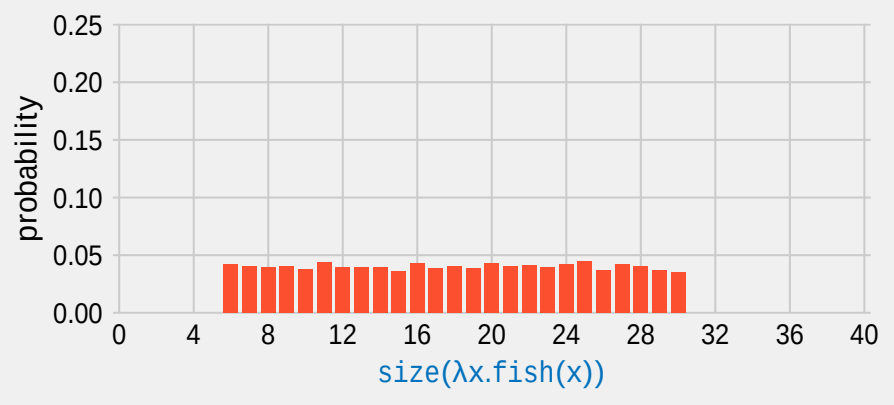


Figure 23: Histogram of the size of the set of fish (i.e. the number of fish), from the MH samples of the theory, after reading the sentences "There are 30 red or blue things," "Every fish is red or blue," and "There are six red fish."

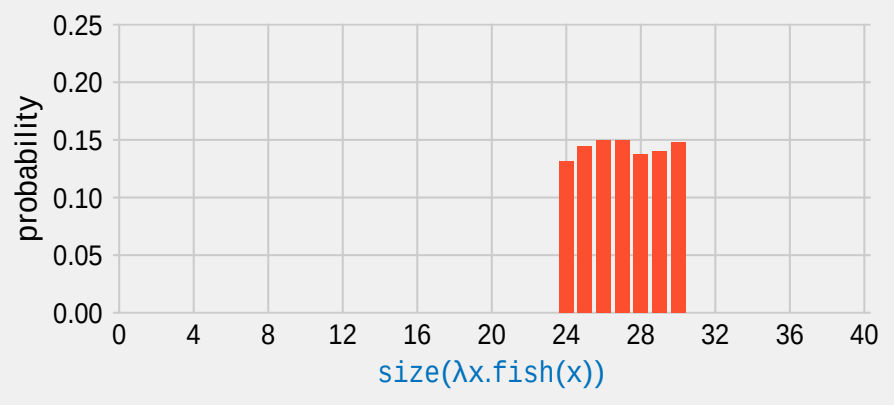


Figure 24: Histogram of the size of the set of fish (i.e. the number of fish), from the MH samples of the theory, after reading the sentences "There are 30 red or blue things," "Every fish is red or blue," "There are six red fish," and "There are 24 blue fish."

histogram of these samples is shown in figure 24. Now, the number of fish is bounded between 24 and 30. If all the fish had distinct colors, there would be 30 fish, but if all of the red fish were also blue, there would 24 fish.

PWL then reads the sentence "No fish is red and blue," and adds the most probable logical form to the theory. After 120,000 iterations of MH, the samples of the number of fish are shown in the histogram in figure 25. Since all fish now have distinct colors, the lower bound on the number of fish is 30, and therefore, the number of fish is fixed to 30.

However, our specialized data structure for checking the consistency of set sizes (described in section 3.2.1.1) does not detect all inconsistencies. We demonstrate this in the above example by replacing the sentence "There are 30 red or blue things," with "There are 35 red

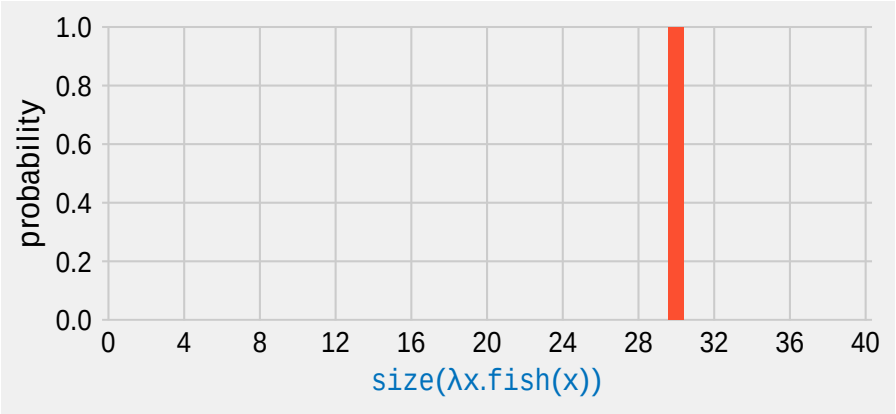


Figure 25: Histogram of the size of the set of fish (i.e. the number of fish), from the MH samples of the theory, after reading the sentences "There are 30 red or blue things," "Every fish is red or blue," "There are six red fish," "There are 24 blue fish," and "No fish is red and blue."

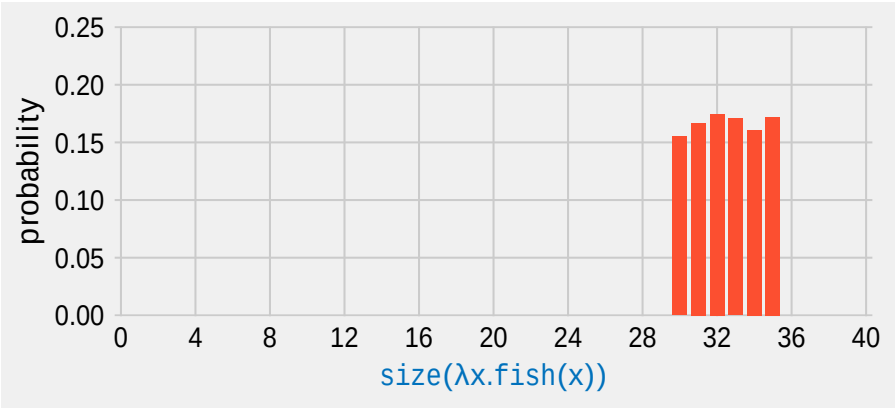


Figure 26: Histogram of the size of the set of fish (i.e. the number of fish), from the MH samples of the theory, after reading the sentences "There are 35 red or blue things," "Every fish is red or blue," "There are six red fish," "There are 24 blue fish," and "No fish is red and blue."

or blue things." PWL reads this sentence in addition to the 4 other sentences mentioned above, adds their most probable logical forms to the theory, and runs 120,000 iterations of MH. The resulting histogram of the number of fish is shown in figure 26. The size of the set of all fish is bounded between 30 and 35. However, it is impossible to have more than 30 fish, since all fish are either red or blue, and there are 6 red fish and 24 blue fish. Our data structure does not currently detect this inconsistency.

5.3 QUESTION-ANSWERING IN PROOFWRITER

In order to quantitatively demonstrate our implementation as a proof-of-concept, we evaluate it on two question-answering tasks. The first is the PROOFWRITER dataset (Tafjord, Dalvi, and Clark, 2021), which itself

is based on the earlier RuleTaker dataset (Clark, Tafjord, and Richardson, 2020). This dataset contains a large number of paragraphs, each followed by a true/false question, and a label indicating the answer. Each paragraph describes a small self-contained situation, and the task is to determine whether the question is true or false, given the situation in the paragraph. An example from the dataset is shown in figure 27. The ProofWriter dataset comes in two versions: one which makes the closed-world assumption, and one that does not. In the closed-world version, if a fact is not provable from its premises, it is false. As a result, all questions are labeled either true or false. In the "open-world" version, some facts are not provably true or false given the premises. These examples are labeled unknown. The dataset is divided into a number of sections, which the authors used to evaluate different aspects of their method. To evaluate and demonstrate the out-of-domain language understanding and reasoning ability of PWL, we use the Birds-Electricity "open-world" portion of the dataset, as the authors evaluated their method on this portion with the same goal. They evaluated their method on this portion by measuring its *zero-shot* accuracy, where the algorithm is evaluated without being trained on examples from the same dataset. We also evaluate PWL by measuring zero-shot accuracy. This portion of the data is further subdivided into 6 sections, each with varying degrees of difficulty.

For each example in the Birds-Electricity portion of the dataset, PWL reads the context and abduces a theory using MH. Next, it parses the query sentence y into a logical form x and estimates its *unnormalized* probability $p(x \mid x_1, \ldots, x_n)$ using the sampling method described in section 3.3.1. To approximate this quantity, MH is run for 400 iterations, and at every 100^{th} iteration, PWL re-initializes the Markov chain by performing 20 "exploratory" MH steps (i.e. a random walk, where MH only considers the third and fourth proposals in table 1 and accepting every proposal). Once PWL has computed this probability for the query sentence, it does the same for the negation of the sentence. These unnormalized probabilities are then compared, and if they are within 2000 in log probability, PWL returns the label unknown. If the first probability is sufficiently larger than the second, PWL returns true, and otherwise, return false. The parameters in the prior were set by hand initially by choosing values which we thought were reasonable (e.g. the average length of a natural deduction proof for a sentence containing a simple subject noun phrase, object noun phrase, and transitive verb is around 20 steps, which is why the Poisson parameter for the proof length is set to 20). The values were tweaked as necessary by running the algorithm on toy examples during debugging. Note that the sentences "Bill is a bird" and "Bill is not a bird" can still both be true if each "Bill" refers to distinct entities. To avoid this, we chose an extreme value of the prior parameter such that the log prior probability of a theory with two entities having the same name is 2000 less than

Context: "Arthur is a bird. Arthur is not wounded. Bill is an ostrich. Colin is a bird. Colin is wounded. Dave is not an ostrich. Dave is wounded. Every ostrich is a bird. Every ostrich is abnormal. Every ostrich is not flying. Every bird that is wounded is abnormal. Every thing that is wounded is not flying. Every bird that is not abnormal is flying."

Query: "Bill is a bird." **true**, **false**, **unknown?**

Figure 27: An example from the Birds1 section in the PROOFWRITER dataset. Its label is true.

Context: "The switch is on. The circuit has the bell. If the circuit has the switch and the switch is on then the circuit is complete. If the circuit does not have the switch then the circuit is complete. If the circuit is complete and the circuit has the light bulb then the light bulb is glowing. If the circuit is complete and the circuit has the bell then the bell is ringing. If the circuit is complete and the circuit has the radio then the radio is playing."

Query: "The circuit is complete." **true**, **false**, **unknown?**

Figure 28: Another example from the Electricity1 section in the PROOFWRITER dataset. Its label is unknown. However, under classical logic, the query is provably true from the information in the 1st, 3rd, and 4th sentences. This is not typical; classical and intuitionistic logic produce the same result for most examples in the PROOFWRITER dataset.

that of a theory where the name is unique. It is for this reason 2000 was chosen as the threshold for determining whether a query is true/false vs unknown. This prior worked well enough for the experiments in this thesis, but the goal is to have a single prior work well for any task, so further work to explore which priors work better across a wider variety of tasks is welcome. We evaluated PWL on PROOFWRITER using both classical and intuitionistic logic, even though the ground truth labels in the dataset were generated using intuitionistic logic. Figure 28 showcases an example where the provability of the question is dependent on the choice of logic.

Table 3 lists the zero-shot accuracy of PWL, comparing with baselines based on the T5 transformer (Raffel et al., 2020). The baseline methods are black-box models that are trained to take, as input, the paragraph and question, and output a proof of whether the question is true or false (or "None" if there is no proof). The baseline ProofWriter-All feeds the input into a T5 transformer and outputs this proof all-at-once. On the other hand, ProofWriter-Iter performs the task iteratively: it feeds the input into a T5 transformer and outputs a *single proof step*. The conclusion of this step is then added to the input and this process is repeated: the new (larger) input is fed into the T5 transformer, which again outputs a proof step. The process is repeated until there are no

further proof steps available. The final proof can then be constructed from the individual proof steps. The baseline ProofWriter-All is trained on the D5 portion of the ProofWriter dataset, whereas ProofWriter-Iter is trained on the D0, D1, D2, and D3 portions.

We emphasize here that PWL is not perfectly comparable to the baselines, since they aimed to demonstrate that their method can *learn* to reason. But we chose to compare against them since they were the only available baselines on the dataset . We instead aim to demonstrate that PWL's ability to parse and reason end-to-end generalizes to an out-of-domain question-answering task. The baseline is trained on other portions of the ProofWriter data, whereas the parser in PWL is trained only on its seed training set and the reasoning module is not trained. PWL performed much better using intuitionistic logic than classical logic, as expected since the ground truth labels were generated using intuitionistic semantics. However, most humans and real-world reasoning tasks would take the law of the excluded middle to be true, and classical logic would serve as a better default. Although the task is relatively simple, it nevertheless demonstrates the proof-of-concept and the promise of further research in this program.

Section	N	ProofWriter-All	ProofWriter-Iter	PWL (classical)	(intuitionistic)
Electricity1	162	98.15	98.77	92.59	**100.00**
Electricity2	180	91.11	90.00	90.00	**100.00**
Electricity3	624	91.99	94.55	88.46	**100.00**
Electricity4	4224	91.64	99.91	94.22	**100.00**
Birds1	40	**100.00**	95.00	**100.00**	**100.00**
Birds2	40	**100.00**	95.00	**100.00**	**100.00**
Average	5270	91.99	98.82	93.43	**100.00**

Table 3: Zero-shot accuracy of PWL and baselines on the ProofWriter dataset.

5.4 QUESTION-ANSWERING IN FICTIONALGEOQA

The sentences in the ProofWriter experiment are template-generated and have simple semantics, and therefore, they do not provide a good representation of real-world NLU. For the sake of a more representative evaluation, we introduce a new question-answering dataset called FictionalGeoQA. FictionalGeoQA is a dataset containing paragraphs, each followed by a question and a label indicating the correct answer(s). Unlike ProofWriter, the questions in FictionalGeoQA are *not* true/-false. For many questions, the correct answer is a collection of multiple entities rather than a single entity, such as in "What rivers in Wulstershire are not major?" For some questions, the correct answer is the empty set (e.g. "What rivers in Wulstershire are not major?" but no river in Wulstershire is major).

To create this dataset, we took questions from GeoQuery (Zelle and Mooney, 1996), and for each question, we wrote a paragraph context containing the information necessary to answer the question. We added distractor sentences to make the task more robust against heuristics. For example, a common heuristic for answering questions about superlatives ("What is the tallest...", etc) is to find the largest number in the paragraph and return the object associated with it. So for each of the questions involving superlatives, we added distractor sentences that contained numbers larger than the other numbers in the paragraph. Whenever possible, the sentences in this paragraph were taken from Simple English Wikipedia. However, some facts, such as the lengths of rivers, are not expressed in sentences in Wikipedia (they typically appear in a table on the right side of the page), and so we wrote those sentences by hand. To keep the focus of the evaluation on reasoning ability and to avoid reducing the evaluation to a semantic parsing benchmark, we chose to restrict the complexity of the language. In particular, each sentence is independent and can be understood in isolation (e.g. there are no cross-sentential anaphora). The sentences *are* more complex than those in ProofWriter, having more of the complexities of real language, such as synonymy, lexical ambiguity (e.g. what is the semantics of "has" in "a state has city" vs "a state has area"; or whether "largest state" refers to area or population), and syntactic ambiguity. This dataset also contains sentences with more complex semantics, such as definitions of new concepts. For example, there are examples in the dataset where the concept "major" is defined as "Every river that is longer than 400 kilometers is major." Reading this definition and then reasoning about it is required in order to correctly answer the question "What rivers in Wulstershire are major?" This increased difficulty of the dataset is evident in the results. We replaced all place names with fictional names in order to remove any confounding effects from pretraining. The dataset consists of 600 paragraph-question pairs and is freely available in our repository. This dataset is meant to evaluate out-of-domain generalizability, and so we do not provide a separate training set for fine-tuning. However, this presents a problem when methods output the correct answer for a question, but phrased slightly differently from the label in the dataset. This is most apparent when language models overgenerate text. One way to workaround this problems it to evaluate the answers manually, but this is time-consuming. As such, we wrote a Python script to automatically evaluate the outputs of each method, where each question is labeled with a list of simple patterns. If the answer matches any of the patterns, it is considered correct.

We compare PWL (using classical logic) with a number of baselines: (1) UnifiedQA (Khashabi et al., 2020), a question-answering system based on large-scale neural language models, (2) Boxer (Bos, 2015), a wide-coverage symbolic semantic parser, combined with Vampire 4.5.1

Context: "River Giffeleney is a river in Wulstershire. River Wulstershire is a river in the state of Wulstershire. River Elsuir is a river in Wulstershire. The length of River Giffeleney is 413 kilometers. The length of River Wulstershire is 830 kilometers. The length of River Elsuir is 207 kilometers. Every river that is shorter than 400 kilometers is not major."

Query: "What rivers in Wulstershire are not major?"

Figure 29: An example from FictionalGeoQA, a new fictional geography question-answering dataset that we created to evaluate reasoning in natural language understanding.

(Kovács and Voronkov, 2013), a theorem prover for full first-order logic, (3) Boxer combined with E 2.6 (Schulz, Cruanes, and Vukmirovic, 2019), another theorem prover for full first-order logic, (4) the language module of PWL combined with Vampire, and (5) the language module of PWL combined with E. The results are shown in figure 4, along with a breakdown across multiple subsets of the dataset. The difficulty of this task relative to ProofWriter is evident from these results. UnifiedQA performs relatively well but fairs more poorly on questions with negation and subjective concept definitions (e.g. "Every river longer than 500km is major... What are the major rivers?"). Humans are easily able to understand and utilize such definitions, and the ability to do so is instrumental in learning about new concepts or vocabulary in new domains. PWL is able to fare better than UnifiedQA in examples with lexical ambiguity, as a result of the language module's ability to exploit acquired knowledge to resolve ambiguities. We find that Boxer has significantly higher coverage than PWL (100% vs 79.8%) but much lower precision. For instance, Boxer uses the semantic representation in the Parallel Meaning Bank (Abzianidze et al., 2017) which has a simpler representation of superlatives, and is thus unable to capture the correct semantics of superlatives in examples of this dataset. We also find that for most examples, Boxer produces different semantics for the question than for the context sentences, oftentimes predicting the incorrect semantic role for the interrogative words, which leads to the theorem provers being unable to find a proof for these extra semantic roles. We also experimented with replacing our reasoning module with a theorem prover and found that for almost all examples, the search of the theorem prover would explode combinatorially and would timeout. This was due to the fact that our semantic representation relies heavily on *sets*, and so a number of simple set theoretic axioms are required for the theorem provers, but this quickly causes the deduction problem to become undecideable. Our reasoning module instead performs abduction, and is able to create axioms to more quickly find an initial proof, and then refine that proof using MH. Despite our attempt to maximize the generalizability of the grammar in PWL, there are a number of linguistic phenomena that we did not yet

	ALL	SUPERLATIVE	SUBJECTIVE CONCEPT DEF.	OBJECTIVE CONCEPT DEF.	LEXICAL AMBIGUITY	NEGATION	LARGE CONTEXT	ARITHMETIC	COUNTING	0 SUBSETS	1 SUBSET	2 SUBSETS	3 SUBSETS	4 SUBSETS
N	600	210	150	170	180	102	100	20	30	85	213	187	85	30
UnifiedQA	33.8	29.5	7.3	33.5	32.8	14.7	43.0	10.0	**20.0**	41.2	47.9	27.8	8.2	**23.3**
Boxer + E	9.7	0.0	12.0	11.8	0.0	15.7	14.0	10.0	0.0	7.1	17.8	5.3	4.7	0.0
Boxer + Vampire	9.7	0.0	12.0	11.8	0.0	15.7	14.0	10.0	0.0	7.1	17.8	5.3	4.7	0.0
PWL parser + E	5.0	0.0	13.3	2.9	0.0	15.7	4.0	10.0	0.0	1.2	7.0	5.3	4.7	0.0
PWL parser + Vampire	9.0	0.0	13.3	11.2	0.0	15.7	4.0	10.0	0.0	12.9	13.6	5.3	4.7	0.0
PWL	**43.1**	**40.5**	**33.3**	33.5	**34.4**	**23.5**	**45.0**	10.0	0.0	**43.5**	**62.9**	**39.0**	**17.6**	0.0

LEGEND:

- **SUPERLATIVE** The subset of the dataset with examples that require reasoning over superlatives, i.e. "longest river."
- **SUBJECTIVE CONCEPT DEF.** Subset with definitions of "subjective" concepts, i.e. "Every river longer than 500 km is major."
- **OBJECTIVE CONCEPT DEF.** Subset with definitions of "objective" concepts, i.e. the population of a location is the number of people living there.
- **LEXICAL AMBIGUITY** Subset with lexical ambiguity, i.e. "has" means different things in "a state has a city named" vs "a state has an area of..."
- **NEGATION** Subset with examples that require reasoning with classical negation (negation-as-failure is insufficient).
- **LARGE CONTEXT** Subset of examples where there are at least 100 sentences in the context.
- **ARITHMETIC** Subset with examples that require simple arithmetic.
- **COUNTING** Subset with examples that require counting.
- n **SUBSET(S)** Examples that belong to exactly n of the above subsets (no example is a member of more than 4 subsets).

Table 4: Zero-shot accuracy of PWL and baselines on the FICTIONALGEOQA dataset.

implement, such as interrogative subordinate clauses, wh-movement, spelling or grammatical mistakes, etc, and this led to the lower coverage on this dataset. Work remains to be done to implement these missing production rules in order to further increase the coverage of the parser.

Each question in the FICTIONALGEOQA dataset is labeled with a list of flags that indicate various features of that question. These flags serve to divide the dataset into the subsets that are shown in figure 4. For example, if a question both requires reasoning with superlatives and has lexical ambiguity, it would be labeled with the flags `superlative` and `lexical_ambiguity`, and would belong to both of those respective sets. This helps to identify how well each method performs on questions with superlatives, lexical ambiguity, negation, etc.

Below, we will provide examples of questions that PWL answered correctly and questions that it answered incorrectly:

- A correctly-answered question with the **SUPERLATIVE** flag:

Context: "The Merasardu River is a river in Efanangole. The Merafagole River is a river in Efanangole. The Mbalam is a river in Bolurofi. The Kolufori River is a river in Efanangole. There are 3 rivers in Efanangole. The length of the Merasardu River is 432 kilometers. The length of the Merafagole River is 218 kilometers. The length of the Mbalam River is 1297 kilometers. The length of the Kolufori River is 587 kilometers."

Query: "Which is the longest river in Efanangole?"

- An incorrectly-answered question with the **SUPERLATIVE** flag:

Context: "The Merasardu River is a river in Efanangole. The Merafagole River is a river in Efanangole. The Mbalam is a river in Bolurofi. The Kolufori River is a river in Efanangole. There are 3 rivers in Efanangole. The length of the Merasardu River is 432 kilometers. The length of the Merafagole River is 218 kilometers. The length of the Mbalam River is 1297 kilometers. The length of the Kolufori River is 587 kilometers."

Query: "How long is the longest river in Efanangole?"

The grammar in PWL does not currently implement wh-movement, and so the parser fails to parse the question.

- A correctly-answered question with the **SUBJECTIVE CONCEPT DEF.** flag:

Context: "River Giffeleney is a river in Wulstershire. River Wulstershire is a river in the state of Wulstershire. River Elsuir is a river in Wulstershire. The length of River Giffeleney is 413 kilometers. The length of River Wulstershire is 830 kilometers. The length of River Elsuir is 207 kilometers. Every river that is longer than 400 kilometers is major."

Query: "What are the major rivers in Wulstershire?"

- An incorrectly-answered question with the **SUBJECTIVE CONCEPT DEF.** flag:

Context: "Voronolga is a province in Bievorsk. Abdorostan is a province in Bievorsk. Fordgorod is a province in Bievorsk. There are 7 provinces. Galininograd is a province in Bievorsk. Puotorsk is a province in Bievorsk. Getarovo is a province in Bievorsk. Bripetrsk is a province in Bievorsk. The population of Voronolga is 10178291. The area of Voronolga is 671200 square kilometers. The population of Abdorostan is 6712302. The area of Abdorostan is 912300 square kilometers. The population of Fordgorod is 7290132. The area of Fordgorod is 671300 square kilometers. The population of Galininograd is 2392010. The area of Galininograd is 314900 square kilometers. The population of Puotorsk is 410293. The area of Puotorsk is 1023900 square kilometers. The population of Getarovo is 2912062. The area of Getarovo is 519200 square kilometers. The population of Bripetrsk is 412908. The area of Bripetrsk is 428100 square kilometers. If the area of X is smaller than 500000 square kilometers then X is minor."

Query: "What are the minor provinces?"

The sentence "If the area of X is smaller than 500000 square kilometers then X is minor" is incorrectly parsed. The comparative "smaller" is ambiguous in that it can refer to smaller in area (e.g. the tennis court is smaller than the football field), in population (e.g. the town is smaller than the city), or in value (e.g. the mass of stone is smaller than 1000kg). PWL interpreted "smaller" to mean smaller in area, but the correct interpretation should be smaller in value, since "area of X" refers to a value rather than an object with an area.

- A correctly-answered question with the **OBJECTIVE CONCEPT DEF.** flag:

Context: "If the number of people living in X is Y then the population of X is Y. Every city is in one state. Wilgalway is a city in the state of Wulstershire. 189202 people live in Wilgalway. The elevation of Wilgalway is 32 meters. Elfincaster is a city in Wulstershire. The population of Elfincaster is 87142."

Query: "What is the population of Wilgalway, Wulstershire?"

- An incorrectly-answered question with the **OBJECTIVE CONCEPT DEF.** flag:

Context: "Baritolloti is the capital city of Grappulia. Grappulia is a state in Catardinia. Grappulia borders Cartabitan. Regnobenoa is a state in Catardinia. Bascilitina is a state in Catardinia. Regnobenoa borders Grappulia. Bascilitina borders Regnobenoa. Bascilitina borders Grappulia. Geoturin is the capital city of the state of Cartabitan. Veronizzia is a city in Cartabitan. Baritolloti is a city in Grappulia. Brescitabia is a city in Cartabitan. Cartabitan is a state in Catardinia. The Begliomento is a river in Catardinia. The Begliomento runs through Grappulia. The Begliomento runs through Regnobenoa. The Terravipacco is a river in Catardinia. The Terravipacco runs through Grappulia. The Terravipacco runs through Regnobenoa. The Terravipacco runs through Bascilitina. The Pernatisone is a river in Catardinia. The Pernatisone runs through Grappulia. If X borders Y, then Y borders X."

Query: "Which rivers run through states that border the state with the capital Baritolloti?"

The grammar in PWL does not currently cover compound noun-noun phrases such as "capital Baritolloti," and so the parser misinterprets the question and provides the reasoning module with a mistaken logical form.

- A correctly-answered question with the LEXICAL AMBIGUITY flag:

Context: "Voronolga is a province in Bievorsk. Abdorostan is a province in Bievorsk. Fordgorod is one of the 7 provinces in Bievorsk. Galininograd is a province in Bievorsk. Puotorsk is a province in Bievorsk. Getarovo is a province in Bievorsk. Bripetrsk is a province in Bievorsk. The area of Voronolga is 671200 square kilometers. The area of Abdorostan is 912300 square kilometers. The area of Fordgorod is 671300 square kilometers. The area of Galininograd is 314900 square kilometers. The area of Puotorsk is 1023900 square kilometers. The area of Getarovo is 519200 square kilometers. The area of Bripetrsk is 428100 square kilometers. There are 7 provinces."

Query: "Puotorsk is how large?"

- Incorrectly-answered question with the LEXICAL AMBIGUITY flag:

Context: "Baritolloti is the capital city of Grappulia. Grappulia is a state in Catardinia. Regnobenoa is a state in Catardinia. The capital of Regnobenoa is Messinitina. Messinitina is a city in Regnobenoa. Lomberona is a city in Regnobenoa. Bascilitina is a state in Catardinia. Lomtrieste is a city in Bascilitina. Albucapua is the capital city of Bascilitina. Geoturin is the capital city of the state of Cartabitan. Veronizzia is a city in Cartabitan. Brescitabia is a city in Cartabitan."

Query: "What is the capital of states that have cities named Baritolloti?"

The grammar in PWL does not currently support passive verb phrase adjuncts of nouns where the verb phrase doesn't contain a "by" indicating the subject. So for example, PWL can correctly

interpret "the state bordered by Canada," it cannot currently parse "the state named Alaska."

- A correctly-answered question with the NEGATION flag:

Context: "Grappulia is a state in Catardinia. Regnobenoa is a state in Catardinia. Bascilitina is a state in Catardinia. Geoturin is the capital city of the state of Cartabitan. Veronizzia is a city in Cartabitan. Baritolloti is a city in Grappulia. Brescitabia is a city in Cartabitan. Cartabitan is a state in Catardinia. The Begliomento is a river in Catardinia. The Begliomento does not run through Grappulia. The Begliomento does not run through Regnobenoa. The Terravipacco is a river in Catardinia. The Terravipacco does not run through Grappulia. The Terravipacco does not run through Regnobenoa. The Terravipacco does not run through Bascilitina. The Pernatisone is a river in Catardinia. The Pernatisone does not run through Grappulia."

Query: "What rivers do not run through Grappulia?"

- An incorrectly-answered question with the NEGATION flag:

Context: "Grappulia is a state in Catardinia. Regnobenoa is a state in Catardinia. Bascilitina is a state in Catardinia. Geoturin is the capital city of the state of Cartabitan. Veronizzia is a city in Cartabitan. Baritolloti is a city in Grappulia. Brescitabia is a city in Cartabitan. Cartabitan is a state in Catardinia. The Begliomento is a river in Catardinia. The Begliomento does not run through Grappulia. The Begliomento does not run through Regnobenoa. The Terravipacco is a river in Catardinia. The Terravipacco does not run through Grappulia. The Terravipacco does not run through Regnobenoa. The Terravipacco does not run through Bascilitina. The Pernatisone is a river in Catardinia. The Pernatisone does not run through Grappulia."

Query: "What rivers do not run through Cartabitan?"

The correct answer should be the empty set, as there is no river that provably runs through Cartabitan. However, PWL only outputs the empty set as an answer if it cannot find any other possible answers. But PWL was able to construct theories in which the Begliomento, Pernatisone, and Terravipacco all ran through Cartabitan. As a result, it did not output the empty set.

- A correctly-answered question with the ARITHMETIC flag:

Context: "Kangoyaken is a state in Gyoshoru. Gunmaishyu is one of the 4 states in Gyoshoru. Ibarakishyo is a state in Gyoshoru. Toyusuma is a state in Dogoreoku. Senkuoka is a state in Gyoshoru. The area of Kangoyaken is 3507 square kilometers. The area of Senkuoka is 4216 square kilometers. The area of Toyusuma is 4198 square kilometers. The area of Ibarakishyo is 9108 square kilometers. The area of Gunmaishyu is 82987 square kilometers. The population of Gunmaishyu is 5218607. The population of Senkuoka is 1027862. The population of Ibarakishyo is 419272. The population of Kangoyaken is 89703. The population of Toyusuma is 19272. The population density of X is the population of X divided by the area of X. If the population density of X is greater than 300, X is dense."

Query: "What are the dense states in Gyoshoru?"

The reasoning module of PWL does not currently support arithmetic, and so it output nothing. However, the correct answer happened to be the empty set, since no states are "dense" according to the definition in the context. In fact, this is the only example with arithmetic that *any method* answered correctly, and they all did so for the same reason.

- An incorrectly-answered question with the ARITHMETIC flag:

Context: "Kangoyaken is a state in Gyoshoru. Gunmaishyu is one of the 4 states in Gyoshoru. Ibarakishyo is a state in Gyoshoru. Toyusuma is a state in Dogoreoku. Senkuoka is a state in Gyoshoru. The area of Kangoyaken is 3507 square kilometers. The area of Senkuoka is 4216 square kilometers. The area of Toyusuma is 4198 square kilometers. The area of Ibarakishyo is 9108 square kilometers. The area of Gunmaishyu is 82987 square kilometers. The population of Gunmaishyu is 5218607. The population of Senkuoka is 1027862. The population of Ibarakishyo is 419272. The population of Kangoyaken is 89703. The population of Toyusuma is 19272. The population density of X is the population of X divided by the area of X."

Query: "What state in Gyoshoru has the lowest population density?"

The reasoning module of PWL does not currently support arithmetic.

- PWL does not correctly answer any questions with the COUNTING flag.
- An incorrectly-answered question with the COUNTING flag:

Context: "Kangoyaken is a state in Gyoshoru. Koruhashi is a city in Kangoyaken. The population of Koruhashi is 132902. Dogayashi is a city in Kangoyaken. The population of Dogayashi is 1923012. Kyoukashino is a city in Kangoyaken. The population of Kyoukashino is 210392. Agarikoshi is a city in Kangoyaken. The population of Agarikoshi is 42910. Kagenegawa is a city in Kangoyaken. The population of Kagenegawa is 813729. Senkuoka is a state in Gyoshoru. There are 2 states. Yotsuyamashi is a city in Senkuoka. The population of Yotsuyamashi is 21390162. Sennouhama is a city in Senkuoka. The population of Sennouhama is 29104. If the population of X is greater than 20000000, X is a metropolis."

Query: "Which state has the fewest metropolises?"

PWL did not correctly interpret "has" as containment (i.e. X is a city in Y should mean that Y has X).

We choose to omit examples with the **LARGE CONTEXT** flag for brevity. PWL was able to answer many such examples correctly, which demonstrates that PWL can scale to examples with more than 100 sentences. However, PWL did become noticeably slower and further work is needed to improve its scalability to much larger inputs. We discuss this in the next chapter and provide suggestions for how scalability can be further improved.

5.5 SUMMARY

In this chapter, we provided qualitative and quantitative results of PWL reading sentences and reasoning about them end-to-end. The qualitative examples at the beginning of the chapter demonstrated that PWL is able to resolve syntactic ambiguities by utilizing acquired knowledge from previously-read sentences. Specifically, we showcased examples where PWL resolves prepositional phrase attachment ambiguity, ambiguity in pronominal resolution, as well as lexical ambiguity. But we also presented examples that highlight current shortcomings of PWL, such as from the overly-simplified nature of the prior on the theory and proofs. We also demonstrated that PWL is capable of discrete reasoning (e.g. counting) via reasoning about the sizes of sets. Finally, we provided quantitative results on two question-answering datasets: PROOFWRITER and a new dataset we called FICTIONALGEOQA. PWL was able to outperform current state-of-the-art baselines on these datasets. We refer the author to the next chapter for more detailed discussion on general conclusions and future work.

6

CONCLUSIONS AND FUTURE WORK

In this thesis, we introduced the *Probabilistic Worldbuilding Model* (PWM), a fully symbolic Bayesian model of semantic parsing and reasoning, which we hope serves as a compelling first step in a research program toward more domain- and task-general natural language understanding. PWL explicitly builds an internal mental model, called the *theory*, akin to the mental model that humans construct when making sense of their observations. We believe that this sort of "worldbuilding" is instrumental in building natural language understanding and AI systems that are able to generalize to new tasks and domains as humans do. We derived *Probabilistic Worldbuilding with Language* (PWL), an efficient inference algorithm that reads sentences by parsing and abducing updates to its latent world model that capture the semantics of those sentences. We demonstrated its ability to exploit acquired knowledge to resolve syntactic ambiguities, such as prepositional phrase attachment and pronominal resolution. We also empirically demonstrated its ability to generalize to two out-of-domain question-answering tasks. In doing so, we created a new question-answering dataset, FictionalGeoQA, designed specifically to evaluate reasoning ability while capturing more of the complexities of real language and being robust against heuristic strategies.

6.1 HIGH-LEVEL CONCLUSIONS

In contrast with past deductive reasoning approaches, PWL instead performs abduction, which is computationally easier, since it can create new axioms *as needed* to find a proof of an input logical form. The highly underspecified nature of the problem of abduction is alleviated by the probabilistic nature of PWL, as it provides a principled way to find the most probable theories.

The probabilistic nature of both the model and inference also helps to remedy the brittleness that plagued fully symbolic deterministic systems. Such systems run into an impasse when they encounter observations or sentences that are inconsistent with the theory. However, in PWL, the theory is random, and so if a given observation is inconsistent with one possible theory, there exist other theories in which the observation is consistent, and the probability distribution over theories provides a principled way to find such consistent theories. PWL is a Bayesian model, where every random variable in the model has a prior distribution. These prior distributions enabled us to incorporate background/expert knowledge into the design of the model, improv-

ing statistical efficiency. For example, we were able to incorporate a great deal of information about English syntax into the grammar of PWL, and as a result, a small seed training set was sufficient to achieve high accuracy in semantic parsing.

We chose a single unified human-readable formal language, higher-order logic, to represent all knowledge in the theory, the intermediate proof steps, as well as the semantics of natural language sentences. Higher-order logic is well-studied and highly expressive, capable of representing the meaning of a very broad class of natural language sentences, and many different kinds of knowledge. This choice helps to keep the theory unspecialized to any particular domain, task, or modality, more similar to the internal mental model in human language understanding. In addition, the expressivity allows PWL to read and understand sentences with richer semantics, such as definitions of previously unknown concepts. Future work to evaluate PWL on other modalities and tasks would interesting. PWL could be extended to other modalities, for example by creating a new "vision module" which would allow semantic information to be extracted from images and incorporated into the theory. Such a module would also enable the flow of information in the reverse direction: to help improve image understanding by incorporating information from previously acquired knowledge. Overall, the modular architecture (i.e. dividing the overall model into the language and reasoning modules) was very useful in the implementation of PWL, as it provided us with the ability to test and debug each module independently.

6.2 REASONING MODULE: CONCLUSIONS AND FUTURE WORK

We chose to use *natural deduction* for the proofs in the reasoning module, a well-studied proof calculus for higher-order logic. And among the well-studied proof calculi for higher-order logic, the deduction rules in natural deduction are most similar to human deductive reasoning; hence the name. Natural deduction is also *complete* in that for any true logical form , there exists a proof of in natural deduction.

During inference, PWL aims to approximate the full posterior distribution of the latent random variables (logical form of each sentence x_i, proof of each logical form $_i$, and the theory T), conditioned on the observations (the sentences y_i). We chose to approximate the full posterior, rather than obtain a point estimate, in order to preserve information about uncertainty. This strategy helps to avoid the brittleness of fully symbolic systems. Furthermore, human languages contain words that express uncertainty, such as "probably," "maybe," "could," etc, which indicates that human language generation and processing preserves information about uncertainty.

However, in this thesis, we made the simplifying assumption that the sentences do not express modality or uncertainty, and so each indi-

vidual sample of T is deterministic. All examples in our experiments are deterministic: no sentence in the datasets contains words that express uncertainty. This is clearly an unrealistic assumption, and to relax it, PWM needs to be extended so that each individual sample of T is probabilistic. If this were the case, PWM would be able to properly define logical forms that express the probability of events, such as the logical form meaning "the probability of 'the cat is sleeping' is 60%." PWM would be able to define "probably" as having probability greater than 50%, and therefore PWL would be able to correctly read and understand the sentence "The cat is probably sleeping."

PWL uses Metropolis-Hastings (MH) to approximate the posterior distribution of the theory T and proofs ¡. MH is able to compute samples from the posterior while avoiding the computation of expensive normalization terms, since they cancel out in the expression for the acceptance probability (see section 2.1). We presented an inference strategy where MH is performed in a streaming fashion, on each sentence, where the previous sample of the theory and proofs provides a warm-start for inference of the next sentence, reducing the number of MH iterations needed to find a good approximation of the posterior theory and proofs. In principle, with enough iterations, a single Markov chain can provide samples from the full posterior of the theory and proofs of each logical form. However, in settings with high uncertainty, the true posterior may be highly multi-modal, in which case using multiple Markov chains would be a better approach, and would require fewer iterations of MH to obtain representative samples of the true posterior. MH is not the only MCMC method that can provide posterior samples efficiently, and there may be other MCMC methods that work similarly well.

We made the simplifying assumption that the posterior for the logical forms x_i is unimodal (i.e. the posterior probability is concentrated in a single logical form). This allowed us to compute the most probable logical form for each sentence and fix that logical form as the point estimate of the posterior. Thus the logical form does not vary in subsequent inference. However, the posterior for the logical forms is not always unimodal, even in consideration of the context and background information. Relaxing this assumption would enable PWL to handle scenarios in which there is greater uncertainty in the logical forms.

Some priors in PWM were chosen for ease of implementation and rapid prototyping, such as the prior on the theory p(T) and the proofs p(¡ j T). Recall that the theory T is a collection of axioms $f a_1, a_2, :::g$ and the prior for the theory p(T) generates these axioms sequentially, conditioned on the fact that the axioms are not inconsistent with one another. This prior has some desirable properties: such as the fact that simpler theories have higher probability than more complex ones (Occam's razor) and that inconsistent theories have zero probability. But the prior is fairly unstructured, and a promising direction for future

work would be to explore more structured priors, such as those that explicitly generate a hierarchical ontology of types. The prior for proofs $p(\ _i\ j\ T)$ is also overly simplified: the premises of each proof step are chosen to be uniformly distributed from the conclusions of previous proof steps. In contrast, human reasoning is likely much more directed, often relying on repeating proofs patterns that appear across many different proofs. For example, PWL often used a "proof by exclusion" pattern in its consistency checking: If a set is known to have size n, and has provable elements $\{x_1, \ldots, x_n\}$, and if a logical form A being true would imply the existence of a new provable element x_{n+1}, then we can conclude that A is false. This proof pattern consists of multiple proof steps in natural deduction. And despite this proof pattern being used repeatedly, PWM assumes the pattern is generated independently each time it is used. A better prior would give higher probability to proof patterns that have been used multiple times in previous proofs. Constructing a prior distribution for proofs that include these features would be valuable for future work.

Perhaps the most consequential assumption in this thesis is that every sentence is conditionally independent of every other sentence, given the theory. Much of real-world natural language violates this assumption, and phenomena such as cross-sentential anaphora would be impossible under this assumption. A proper model of context is necessary to relax this assumption, where the generative process of a logical form is influenced by previous logical forms. In section 4.7, we provide suggestions for a model of context, where the context keeps track of the current topic, the universe of discourse (including discourse narrowing/widening), and recently-mentioned entities which can be used to generate longer-range and cross-sentential anaphora.

In addition, there are many research questions on the issue of *scalability*. Although PWL is able to scale to examples in FictionalGeoQA with more than 100 sentences, there are two main bottlenecks currently preventing the model from scaling to significantly larger theories: (1) the maintenance of global consistency, and (2) the unfocused nature of the current MH proposals. When checking for consistency of a new axiom, rather than considering all other axioms/sets in the theory, it would be preferable to only consider the portion of the theory relevant to the new axiom. But to do so is not obvious. How would we define "relevance"? Relaxing global consistency would allow theory samples to be inconsistent. Would this inconsistency be due to the approximate nature of inference (i.e. the samples are approximating an underlying consistent theory)? Or is the inconsistency a part of the model (in the underlying theory itself)? If the latter, PWM must be modified to generate inconsistencies. Additionally, the current MH proposals do not take into account the goal of reasoning. For example, if the current task is to answer a question about geography, then MH proposals for proofs unrelated to geography are wasteful, and would increase the

number of MH steps necessary to sufficiently mix the Markov chain. A more clever goal-aware approach for selecting proofs to mutate would help to alleviate this problem and improve scalability.

6.3 LANGUAGE MODULE: CONCLUSIONS AND FUTURE WORK

The model of the language module is an extension of a context-free grammar (CFG). In any derivation tree (i.e. syntax tree), every node is associated with a logical form which represents the meaning of the corresponding fragment of the sentence. For each production rule in the grammar, every right-hand side nonterminal is associated with a semantic transformation function. These transformation functions characterize the relationship between the logical form of the parent node and that of the child node. Our training procedure presented in section 4.3.1 induces preterminal production rules (i.e. rules of the form N ! "tennis"). However, the other production rules in the grammar were specified by hand. This gave us a high level of control over the grammar, but was also very time-consuming. An interesting avenue for future research would be to explore how these production rules could be induced, as well. In section 4.7, we suggest one approach where semantic transformation functions can be written as short "programs" in a simple programming language (i.e. a sequence of instructions). While grammar induction is interesting in its own right, and would save a great deal of time when writing a new grammar, such as for languages other than English, there is no obvious reason to believe that it would meaningfully improve parsing accuracy.

In the generative process for the derivation trees, PWL uses hierarchical Dirichlet processes (HDPs) to model the distribution of production rules. We presented a novel application of HDPs where distributions can depend on discrete structures, such as logical forms, and this dependence can be learned from data (see section 4.1.4). More specifically, the HDP in PWL defines a distribution over the production rules that make up the derivation trees. Our training procedure in section 4.3.1 learns the relationship between the logical forms and the distribution of these production rules. Each level of the HDP corresponds to dependence on a specific feature of the logical form (e.g. the predicate of an atom, or the value of a constant in the left-most conjunct, etc), and so additional levels can be added to the hierarchy in order to add additional dependence on more features of the logical form. But the HDP is not the only choice to model the distribution of production rules in derivation trees. For instance, the order of the features of the logical form is important when constructing the HDP hierarchy. A different order of features will produce a different hierarchy. This may be advantageous in some cases. For example if we know that feature f_A provides more information about the distribution of production rules than feature f_B, then this inductive bias can be reflected by associating

the first level of the HDP hierarchy with f_A and associating the second level with f_B. But it would be interesting to explore the use of alternative conditional distributions for the production rules, including those that are independent of the order of the features. Our parser requires an efficient algorithm to compute the k most likely sets of logical forms, given an observed production rule (see sections 4.1.2 and 4.3.2). So in order to use our parser with an alternative conditional distribution, a similar algorithm is required.

We designed and implemented a new broad-coverage semantic formalism and grammar for English in section 4.5. This semantic formalism was built on higher-order logic in order to capture a wide variety of linguistic phenomena and so that the logical forms can be directly used by the reasoning module. In this formalism, events and predicates are represented as existentially-quantified objects (i.e. neo-Davidsonian semantics). Named entities are also represented as existentially-quantified objects, which means the semantic parser is no longer responsible for named entity linking. Instead, the reasoning module resolves named entities. The structure of logical forms are made to closely mirror the syntactic structure of the corresponding sentences or phrases, which helps to simplify semantic parsing. Unlike AMR, our semantic formalism is able to represent richer semantics, such as negation and universal quantification. And unlike DRT, we are able to use well-studied reasoning methods which were developed for first- and higher-order logic. Due to limited time, we did not implement a number of core features of the English language into the new grammar, such as interrogative subordinate clauses, wh-movement, imperative mood, and others. But extending the grammar to include these features is not difficult in the established framework. Additional production rules/semantic transformation functions can be written to do so.

The semantic representation of sentences also needs to be extended to properly represent modality and intensionality. A promising avenue to do so is to allow quantifiers to quantify over both real and hypothetical objects, and real objects would be specially marked with a new `real` predicate (see section 4.7). Then, our semantic formalism would be able to express statements about both real and hypothetical objects.

In addition, even though PWL is fully symbolic, non-symbolic methods could be used for expressive prior/proposal distributions or approximate inference. For example, the distribution for selecting production rules in the language module could be replaced with a richer prior distribution. The whole language module itself could be replaced with a different model of natural language semantics; perhaps with one that is not based on a grammar. All-in-all, there are many fascinating research paths to pursue from here.

6.4 FUTURE OF NATURAL LANGUAGE UNDERSTANDING

Looking ahead, natural language understanding and artificial intelligence stands to benefit immensely from models with the ability to reason, especially in a domain-, modality-, and task-independent manner. We posit that this reasoning ability, whether trained or built-in, is instrumental in building NLU and AI systems that can generalize to new tasks and domains to the same degree as humans. This ability would also help to address problems like catastrophic forgetting, where a model may be trained to perform well on a task, but when trained on a second task, its performance on the first task deteriorates (the model "forgets" how to do the first task).

Symbolic representations of meaning can be very useful in constructing systems with the aforementioned ability to reason, as they can draw upon the vast work in symbolic reasoning, automated deduction, proof theory, formal semantics, etc. Symbolic representations also facilitate interpretability, which is particularly useful to discern *why* an NLU/AI system behaves the way that it does. This ability will become hugely important when developing and debugging larger systems with multiple interacting components. A promising direction is the recent work into neuro-symbolic methods that endeavor to capture the advantages of symbolic representations. We should also not be so quick to dismiss fully symbolic approaches either. A large diversity of active research directions is an indicator of a healthy field of research.

We must also rethink our approach for evaluating methods in AI and NLP. The prevailing approach of pretraining a model and then fine-tuning it on an evaluation dataset is prone to overfitting, and the resulting evaluation may not reflect the algorithm's true ability. For example, the resulting fine-tuned algorithm may not perform as well on a new similar dataset without fine-tuning again. A shift in focus to out-of-domain evaluation or zero-shot evaluation would help to prioritize the development of more robust algorithms that are not as prone to overfitting. Moving forward, we must be mindful when developing new evaluation datasets to ensure that they are not vulnerable to heuristics, and that they truly do evaluate what they were meant to evaluate.

BIBLIOGRAPHY

Abzianidze, Lasha, Johannes Bjerva, Kilian Evang, Hessel Haagsma, Rik van Noord, Pierre Ludmann, Duc-Duy Nguyen, and Johan Bos (2017). "The Parallel Meaning Bank: Towards a Multilingual Corpus of Translations Annotated with Compositional Meaning Representations." In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers*. Ed. by Mirella Lapata, Phil Blunsom, and Alexander Koller. Association for Computational Linguistics, pp. 242–247.

Aho, Alfred V. and Jeffrey D. Ullman (1972). *The theory of parsing, translation, and compiling. 1: Parsing*. Prentice-Hall. ISBN: 0139145567.

Aldous, David J. (1985). "Exchangeability and related topics." In: *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, pp. 1–198.

Arakelyan, Erik, Daniel Daza, Pasquale Minervini, and Michael Cochez (2021). "Complex Query Answering with Neural Link Predictors." In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider (2013). "Abstract Meaning Representation for Sembanking." In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse, LAW-ID@ACL 2013, August 8-9, 2013, Sofia, Bulgaria*. Ed. by Stefanie Dipper, Maria Liakata, and Antonio Pareja-Lora. The Association for Computer Linguistics, pp. 178–186.

Bellodi, Elena and Fabrizio Riguzzi (2015). "Structure learning of probabilistic logic programs by searching the clause space." In: *Theory Pract. Log. Program.* 15.2, pp. 169–212.

Bender, Emily M. and Alexander Koller (2020). "Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault. Association for Computational Linguistics, pp. 5185–5198.

Bhagavatula, Chandra, Ronan Le Bras, Chaitanya Malaviya, Keisuke Sakaguchi, Ari Holtzman, Hannah Rashkin, Doug Downey, Wentau Yih, and Yejin Choi (2020). "Abductive Commonsense Reasoning." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Blunsom, Phil and Trevor Cohn (2010). "Inducing Synchronous Grammars with Slice Sampling." In: *Human Language Technologies: Confer-*

ence of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA. The Association for Computational Linguistics, pp. 238–241.

Bos, Johan (2015). "Open-Domain Semantic Parsing with Boxer." In: *Proceedings of the 20th Nordic Conference of Computational Linguistics, NODALIDA 2015, Institute of the Lithuanian Language, Vilnius, Lithuania, May 11-13, 2015*. Ed. by Beáta Megyesi. Vol. 109. Linköping Electronic Conference Proceedings. Linköping University Electronic Press / Association for Computational Linguistics, pp. 301–304.

Bron, Coenraad and Joep Kerbosch (1973). "Finding All Cliques of an Undirected Graph (Algorithm 457)." In: *Commun. ACM* 16.9, pp. 575–576.

Brown, Tom B. et al. (2020). "Language Models are Few-Shot Learners." In: *CoRR* abs/2005.14165v4.

Charniak, Eugene and Robert P. Goldman (1993). "A Bayesian Model of Plan Recognition." In: *Artif. Intell.* 64.1, pp. 53–79.

Chomsky, Noam (1956). "Three models for the description of language." In: *IRE Trans. Inf. Theory* 2.3, pp. 113–124.

Church, Alonzo (1940). "A Formulation of the Simple Theory of Types." In: *J. Symb. Log.* 5.2, pp. 56–68.

Clark, Peter, Oyvind Tafjord, and Kyle Richardson (2020). "Transformers as Soft Reasoners over Language." In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, pp. 3882–3890.

Cohn, Trevor, Phil Blunsom, and Sharon Goldwater (2010). "Inducing Tree-Substitution Grammars." In: *J. Mach. Learn. Res.* 11, pp. 3053–3096.

Cooper, Robin, Simon Dobnik, Shalom Lappin, and Staffan Larsson (2015). "Probabilistic Type Theory and Natural Language Semantics." In: *Linguistic Issues in Language Technology, Volume 10, 2015*. CSLI Publications.

Cussens, James (2001). "Parameter Estimation in Stochastic Logic Programs." In: *Mach. Learn.* 44.3, pp. 245–271.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Association for Computational Linguistics, pp. 4171–4186.

Dong, Li and Mirella Lapata (2016). "Language to Logical Form with Neural Attention." In: *CoRR* abs/1601.01280.

Dowty, D. R. (1981). *Introduction to Montague Semantics*. eng. 1st ed. 1981. Studies in Linguistics and Philosophy, 11. Dordrecht: Springer Netherlands. ISBN: 94-009-9065-0.

Dreyfus, H. L. (1985). "From Micro-Worlds to Knowledge Representation: AI at an Impasse." In: *Readings in Knowledge Representation*. Ed. by R. J. Brachman and H. J. Levesque. Los Altos, CA: Kaufmann, pp. 71–93.

Dunietz, Jesse, Gregory Burnham, Akash Bharadwaj, Owen Rambow, Jennifer Chu-Carroll, and David A. Ferrucci (2020). "To Test Machine Comprehension, Start by Defining Comprehension." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault. Association for Computational Linguistics, pp. 7839–7859.

Durrett, Rick (2010). *Probability: Theory and Examples, 4th Edition*. Cambridge University Press. ISBN: 9780511779398.

Earley, Jay (1970). "An Efficient Context-Free Parsing Algorithm." In: *Commun. ACM* 13.2, pp. 94–102.

Escobar, Michael D. and Mike West (June 1995). "Bayesian Density Estimation and Inference Using Mixtures." In: *Journal of the American Statistical Association* 90.430, pp. 577–588.

Ferguson, Thomas S. (1973). "A Bayesian Analysis of Some Nonparametric Problems." In: *The Annals of Statistics* 1.2, pp. 209 –230.

Finkel, Jenny Rose, Christopher D. Manning, and Andrew Y. Ng (2006). "Solving the Problem of Cascading Errors: Approximate Bayesian Inference for Linguistic Annotation Pipelines." In: *EMNLP 2006, Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing, 22-23 July 2006, Sydney, Australia*. Ed. by Dan Jurafsky and Éric Gaussier. ACL, pp. 618–626.

Furbach, Ulrich, Andrew S. Gordon, and Claudia Schon (2015). "Tackling Benchmark Problems of Commonsense Reasoning." In: *Proceedings of the Workshop on Bridging the Gap between Human and Automated Reasoning - A workshop of the 25th International Conference on Automated Deduction (CADE-25), Berlin, Germany, August 1, 2015*. Ed. by Ulrich Furbach and Claudia Schon. Vol. 1412. CEUR Workshop Proceedings. CEUR-WS.org, pp. 47–59.

Gallo, Giorgio, Giustino Longo, and Stefano Pallottino (1993). "Directed Hypergraphs and Applications." In: *Discret. Appl. Math.* 42.2, pp. 177–201.

Gardner, Matt, Jonathan Berant, Hannaneh Hajishirzi, Alon Talmor, and Sewon Min (2019). "On Making Reading Comprehension More Comprehensive." In: *Proceedings of the 2nd Workshop on Machine Reading for Question Answering, MRQA@EMNLP 2019, Hong Kong, China, November 4, 2019*. Ed. by Adam Fisch, Alon Talmor, Robin Jia, Minjoon Seo, Eunsol Choi, and Danqi Chen. Association for Computational Linguistics, pp. 105–112.

Geman, Stuart and Donald Geman (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 6.6, pp. 721–741.

Gentzen, G. (1935). "Untersuchungen über das logische Schließen I." In: *Mathematische Zeitschrift* 39, pp. 176–210.

— (1969). "Investigations into Logical Deduction." In: *The Collected Papers of Gerhard Gentzen*. Ed. by M. E. Szabo. Amsterdam: North-Holland, pp. 68–213.

Gregory, Howard (2015). *Language and Logics: An Introduction to the Logical Foundations of Language*. Edinburgh University Press. ISBN: 9780748691623.

Hastings, W. K. (1970). "Monte Carlo sampling methods using Markov chains and their applications." In: *Biometrika* 57.1, pp. 97–109.

Henkin, Leon (1950). "Completeness in the Theory of Types." In: *J. Symb. Log.* 15.2, pp. 81–91.

Hobbs, Jerry R. (1985). "Ontological Promiscuity." In: *23rd Annual Meeting of the Association for Computational Linguistics, 8-12 July 1985, University of Chicago, Chicago, Illinois, USA, Proceedings*. Ed. by William C. Mann. ACL, pp. 61–69.

Hogan, Aidan et al. (2020). "Knowledge Graphs." In: *CoRR* abs/2003.02320v5.

Huddleston, Rodney and Geoffrey K. Pullum (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.

Jain, Arcchit, Tal Friedman, Ondrej Kuzelka, Guy Van den Broeck, and Luc De Raedt (2019). "Scalable Rule Learning in Probabilistic Knowledge Bases." In: *1st Conference on Automated Knowledge Base Construction, AKBC 2019, Amherst, MA, USA, May 20-22, 2019*.

Johnson, Mark, Thomas L. Griffiths, and Sharon Goldwater (2006). "Adaptor Grammars: A Framework for Specifying Compositional Nonparametric Bayesian Models." In: *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*. Ed. by Bernhard Schölkopf, John C. Platt, and Thomas Hofmann. MIT Press, pp. 641–648.

— (2007). "Bayesian Inference for PCFGs via Markov Chain Monte Carlo." In: *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, April 22-27, 2007, Rochester, New York, USA*. Ed. by Candace L. Sidner, Tanja Schultz, Matthew Stone, and ChengXiang Zhai. The Association for Computational Linguistics, pp. 139–146.

Kamp, Hans and Uwe Reyle (1993). *From Discourse to Logic - Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Vol. 42. Studies in linguistics and philosophy. Springer. ISBN: 978-0-7923-1028-0.

— (1996). "A Calculus for First Order Discourse Representation Structures." In: *J. Log. Lang. Inf.* 5.3/4, pp. 297–348.

Kapanipathi, Pavan et al. (2021). "Leveraging Abstract Meaning Representation for Knowledge Base Question Answering." In: *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*. Ed. by Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli. Vol. ACL/IJCNLP 2021. Findings of ACL. Association for Computational Linguistics, pp. 3884–3894.

Kaplan, Ronald M. and Joan Bresnan (1995). *Lexical-Functional Grammar: A Formal System for Grammatical Representation*.

Khashabi, Daniel, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi (2020). "UnifiedQA: Crossing Format Boundaries With a Single QA System." In: *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Vol. EMNLP 2020. Findings of ACL. Association for Computational Linguistics, pp. 1896–1907.

Klein, Dan and Christopher D. Manning (2001). "Parsing and Hypergraphs." In: *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001), 17-19 October 2001, Beijing, China*. Tsinghua University Press.

— (2003). "A* Parsing: Fast Exact Viterbi Parse Selection." In: *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2003, Edmonton, Canada, May 27 - June 1, 2003*. Ed. by Marti A. Hearst and Mari Ostendorf. The Association for Computational Linguistics.

Kotseruba, Iuliia and John K. Tsotsos (2020). "40 years of cognitive architectures: core cognitive abilities and practical applications." In: *Artif. Intell. Rev.* 53.1, pp. 17–94.

Kovács, Laura and Andrei Voronkov (2013). "First-Order Theorem Proving and Vampire." In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, pp. 1–35.

Kwiatkowski, Tom, Eunsol Choi, Yoav Artzi, and Luke S. Zettlemoyer (2013). "Scaling Semantic Parsers with On-the-Fly Ontology Matching." In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, pp. 1545–1556.

Kwiatkowski, Tom, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman (2010). "Inducing Probabilistic CCG Grammars from Logical Form with Higher-Order Unification." In: *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP 2010, 9-11 October 2010, MIT Stata Center, Massachusetts, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, pp. 1223–1233.

Kwiatkowski, Tom, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman (2011). "Lexical Generalization in CCG Grammar Induction for Semantic Parsing." In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP 2011, 27-31 July 2011, John McIntyre Conference Centre, Edinburgh, UK, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, pp. 1512–1523.

Laird, John E., Allen Newell, and Paul S. Rosenbloom (1987). "SOAR: An Architecture for General Intelligence." In: *Artif. Intell.* 33.1, pp. 1–64.

Lake, Brenden M., Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman (2016). "Building Machines That Learn and Think Like People." In: *CoRR* abs/1604.00289v3.

Land, A. H. and A. G. Doig (July 1960). "An Automatic Method of Solving Discrete Programming Problems." In: *Econometrica* 28.3, p. 497.

Li, Peng, Yang Liu, and Maosong Sun (2013). "An Extended GHKM Algorithm for Inducing Lambda-SCFG." In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by Marie desJardins and Michael L. Littman. AAAI Press.

Liang, Percy, Michael I. Jordan, and Dan Klein (2010). "Type-Based MCMC." In: *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA*. The Association for Computational Linguistics, pp. 573–581.

— (2013). "Learning Dependency-Based Compositional Semantics." In: *Comput. Linguistics* 39.2, pp. 389–446.

Linzen, Tal (2020). "How Can We Accelerate Progress Towards Human-like Linguistic Generalization?" In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault. Association for Computational Linguistics, pp. 5210–5217.

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov (2019). "RoBERTa: A Robustly Optimized BERT Pretraining Approach." In: *CoRR* abs/1907.11692v1.

Luo, Yucen, Alex Beatson, Mohammad Norouzi, Jun Zhu, David Duvenaud, Ryan P. Adams, and Ricky T. Q. Chen (2020). "SUMO: Unbiased Estimation of Log Marginal Probability for Latent Variable Models." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

McCoy, Tom, Ellie Pavlick, and Tal Linzen (2019). "Right for the Wrong Reasons: Diagnosing Syntactic Heuristics in Natural Language In-

ference." In: *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Ed. by Anna Korhonen, David R. Traum, and Lluís Màrquez. Association for Computational Linguistics, pp. 3428–3448.

Meyn, Sean P. and Richard L. Tweedie (1993). *Markov Chains and Stochastic Stability*. Communications and Control Engineering Series. Springer. ISBN: 978-1-4471-3269-1.

Mitchell, Tom M. et al. (2018). "Never-ending learning." In: *Commun. ACM* 61.5, pp. 103–115.

Montague, Richard (1970). "Universal grammar." In: *Theoria* 36.3, pp. 373–398.

— (1973). "The Proper Treatment of Quantification in Ordinary English." In: *Approaches to Natural Language*. Springer Netherlands, pp. 221–242.

— (1974). "English as a Formal Language." In: *Formal Philosophy: Selected Papers of Richard Montague*. Ed. by Richmond H. Thomason. New Haven, London: Yale University Press, pp. 188–222.

Muggleton, Stephen (1991). "Inductive Logic Programming." In: *New Gener. Comput.* 8.4, pp. 295–318.

— (1996). "Stochastic Logic Programs." In: *Advances in Inductive Logic Programming*, pp. 254–264.

Newell, Allen and Herbert A. Simon (1976). "Computer Science as Empirical Inquiry: Symbols and Search." In: *Commun. ACM* 19.3, pp. 113–126.

Nie, Yixin, Yicheng Wang, and Mohit Bansal (2019). "Analyzing Compositionality-Sensitivity of NLI Models." In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 6867–6874.

Niepert, Mathias and Pedro M. Domingos (2015). "Learning and Inference in Tractable Probabilistic Knowledge Bases." In: *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI 2015, July 12-16, 2015, Amsterdam, The Netherlands*. Ed. by Marina Meila and Tom Heskes. AUAI Press, pp. 632–641.

Niepert, Mathias, Christian Meilicke, and Heiner Stuckenschmidt (2012). "Towards Distributed MCMC Inference in Probabilistic Knowledge Bases." In: *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction, AKBC-WEKEX@NAACL-HLT 2012, Montrèal, Canada, June 7-8, 2012*. Ed. by James Fan, Raphael Hoffman, Aditya Kalyanpur, Sebastian Riedel, Fabian M. Suchanek, and Partha Pratim Talukdar. Association for Computational Linguistics, pp. 1–6.

Parsons, Terence (1990). *Events in the Semantics of English*. Cambridge, MA: MIT Press.

Pauls, Adam and Dan Klein (2009). "K-Best A* Parsing." In: *ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2-7 August 2009, Singapore*. Ed. by Keh-Yih Su, Jian Su, and Janyce Wiebe. The Association for Computer Linguistics, pp. 958–966.

Pauls, Adam, Dan Klein, and Chris Quirk (2010). "Top-Down K-Best A* Parsing." In: *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden, Short Papers*. The Association for Computer Linguistics, pp. 200–204.

Pfenning, Frank (2004). *Natural Deduction*. Lecture notes in 15-815 Automated Theorem Proving. URL: https://www.cs.cmu.edu/~fp/courses/atp/handouts/ch2-natded.pdf.

Proudian, Derek and Carl Pollard (1985). "Parsing Head-Driven Phrase Structure Grammar." In: *23rd Annual Meeting of the Association for Computational Linguistics, 8-12 July 1985, University of Chicago, Chicago, Illinois, USA, Proceedings*. Ed. by William C. Mann. ACL, pp. 167–171.

Quine, Willard V. (1956). "Quantifiers and Propositional Attitudes." In: *The Journal of Philosophy* 53.5, pp. 177–187. ISSN: 0022362X.

Rabinovich, Maxim, Mitchell Stern, and Dan Klein (2017). "Abstract Syntax Networks for Code Generation and Semantic Parsing." In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, pp. 1139–1149.

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu (2020). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." In: *J. Mach. Learn. Res.* 21, 140:1–140:67.

Ren, Hongyu, Weihua Hu, and Jure Leskovec (2020). "Query2box: Reasoning over Knowledge Graphs in Vector Space Using Box Embeddings." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Riegel, Ryan et al. (2020). "Logical Neural Networks." In: *CoRR* abs/2006.13155.

Robert, Christian P. and George Casella (2004). *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer. ISBN: 978-1-4419-1939-7.

Rocktäschel, Tim and Sebastian Riedel (2017). "End-to-end Differentiable Proving." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*

2017, December 4-9, 2017, Long Beach, CA, USA. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, pp. 3788–3800.

Rothstein, Susan (Jan. 2010). "Counting, Measuring And The Semantics Of Classifiers." In: *Baltic International Yearbook of Cognition, Logic and Communication* 6.1.

Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education. ISBN: 978-0-13-207148-2.

Saha, Swarnadeep, Sayan Ghosh, Shashank Srivastava, and Mohit Bansal (2020). "PRover: Proof Generation for Interpretable Reasoning over Rules." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*. Ed. by Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu. Association for Computational Linguistics, pp. 122–136.

Sandt, Rob A. van der (1992). "Presupposition Projection as Anaphora Resolution." In: *J. Semant.* 9.4, pp. 333–377.

Saparov, Abulhair, Vijay A. Saraswat, and Tom M. Mitchell (2017). "A Probabilistic Generative Grammar for Semantic Parsing." In: *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017*. Ed. by Roger Levy and Lucia Specia. Association for Computational Linguistics, pp. 248–259.

Sato, Taisuke, Yoshitaka Kameya, and Neng-Fa Zhou (2005). "Generative Modeling with Failure in PRISM." In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, pp. 847–852.

Scha, R. (1981). "Distributive, Collective and Cumulative Quantification." In: *Formal Methods in the Study of Language, Part 2*. Ed. by J. A. G. Groenendijk, T. M. V. Janssen, and M. B. J. Stokhof. Mathematisch Centrum, pp. 483–512.

Schulz, Stephan, Simon Cruanes, and Petar Vukmirovic (2019). "Faster, Higher, Stronger: E 2.3." In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*. Ed. by Pascal Fontaine. Vol. 11716. Lecture Notes in Computer Science. Springer, pp. 495–507.

Steedman, Mark (1997). *Surface structure and interpretation*. Vol. 30. Linguistic inquiry. MIT Press. ISBN: 978-0-262-69193-2.

Sun, Haitian, Andrew O. Arnold, Tania Bedrax-Weiss, Fernando Pereira, and William W. Cohen (2020). "Faithful Embeddings for Knowledge Base Queries." In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Process-*

ing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin.

Tafjord, Oyvind, Bhavana Dalvi, and Peter Clark (2021). "ProofWriter: Generating Implications, Proofs, and Abductive Statements over Natural Language." In: *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*. Ed. by Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli. Vol. ACL/IJCNLP 2021. Findings of ACL. Association for Computational Linguistics, pp. 3621–3634.

Tamari, Ronen, Chen Shani, Tom Hope, Miriam R. L. Petruck, Omri Abend, and Dafna Shahaf (2020). "Language (Re)modelling: Towards Embodied Language Understanding." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault. Association for Computational Linguistics, pp. 6268–6281.

Tang, Lappoon R. and Raymond J. Mooney (2000). "Automated Construction of Database Interfaces: Integrating Statistical and Relational Learning for Semantic Parsing." In: *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, EMNLP 2000, Hong Kong, October 7-8, 2000*. Ed. by Hinrich Schütze and Keh-Yih Su. Association for Computational Linguistics, pp. 133–141.

Teh, Yee Whye, Michael I. Jordan, Matthew J. Beal, and David M. Blei (2006). "Hierarchical Dirichlet Processes." In: *Journal of the American Statistical Association* 101.476, pp. 1566–1581.

Wang, Adrienne, Tom Kwiatkowski, and Luke S. Zettlemoyer (2014). "Morpho-syntactic Lexical Generalization for CCG Semantic Parsing." In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. ACL, pp. 1284–1295.

Wikimedia Foundation (2020). *Wiktionary Data Dumps*. URL: https://dumps.wikimedia.org/enwiktionary/.

Wong, Yuk Wah and Raymond J. Mooney (2006). "Learning for Semantic Parsing with Statistical Machine Translation." In: *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 4-9, 2006, New York, New York, USA*. Ed. by Robert C. Moore, Jeff A. Bilmes, Jennifer Chu-Carroll, and Mark Sanderson. The Association for Computational Linguistics.

— (2007). "Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus." In: *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30,*

2007, Prague, Czech Republic. Ed. by John A. Carroll, Antal van den Bosch, and Annie Zaenen. The Association for Computational Linguistics.

Yang, Zhilin, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le (2019). "XLNet: Generalized Autoregressive Pretraining for Language Understanding." In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, pp. 5754–5764.

Yi, Kexin, Chuang Gan, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B. Tenenbaum (2020). "CLEVRER: Collision Events for Video Representation and Reasoning." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Zelle, John M. and Raymond J. Mooney (1996). "Learning to Parse Database Queries Using Inductive Logic Programming." In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*. Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, pp. 1050–1055.

Zettlemoyer, Luke S. and Michael Collins (2005). "Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars." In: *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*. AUAI Press, pp. 658–666.

— (2007). "Online Learning of Relaxed CCG Grammars for Parsing to Logical Form." In: *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*. Ed. by Jason Eisner. ACL, pp. 678–687.

Zhao, Kai and Liang Huang (2015). "Type-Driven Incremental Semantic Parsing with Polymorphism." In: *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*. Ed. by Rada Mihalcea, Joyce Yue Chai, and Anoop Sarkar. The Association for Computational Linguistics, pp. 1416–1421.