CoTs as Tractable Probabilistic Programs

Kyle Richardson[*]
Allen Institute for Artificial Intelligence

Yu Feng[*]
University of Pennsylvania

Poorva Garg
University of California Los Angeles

Junyan Cheng
Dartmouth College

Guy Van den Broeck
University of California Los Angeles

Dan Roth
University of Pennsylvania

Abstract

Chain-of-thought (CoT) traces are used across language model prompting, training, test-time inference, and interpretability, yet they are often modeled in task-specific ways. We propose treating CoT traces as discrete probabilistic programs and introduce Copper, a language in which reasoning steps are stochastic variables, control flow encodes dependencies, and answers are program outputs. Its finite reachability semantics reduces CoT-related quantities to probabilistic queries over Copper programs, placing CoT analysis within tractable probabilistic inference. This formulation recovers standard trace-likelihood quantities while exposing limitations of likelihood-only scoring. It also supports differentiable operators for composing likelihoods, confidence estimates, and feedback over trace structure. We present the language and its tractable semantics, establish several basic formal correspondences with standard CoT inference, and outline ongoing case studies using the framework to derive new test-time inference strategies and verification techniques.

1 Introduction

Since the seminal work of Wei et al. (2022), chain-of-thought (CoT) prompting has become central to how language models are used, evaluated, and improved. CoT traces now appear not only in prompting, but also in model training (Guo et al., 2025; Yeo et al., 2025; Chen et al., 2025), test-time inference (Huang et al., 2025), and interpretability analysis (Turpin et al., 2023; Lanham et al., 2023; Stechly et al., 2024). Yet these uses often treat traces in task-specific ways, without a shared account of what traces denote, how their steps compose, or what inferences they support.

We argue that CoT traces can be modeled as simple probabilistic programs: ordinary programs extended with stochastic choices, branching, conditioning, and probabilistic queries (Gordon et al., 2014; Katoen et al., 2015; De Raedt & Kimmig, 2015). Figure 1 illustrates the idea. Reasoning steps become stochastic assignments, dependencies among steps become control flow, evidence is incorporated using `observe(.)`, and answers are program outcomes. Questions about a trace then become probabilistic queries: answer probabilities used in learning or inference are computed as differentiable marginals, while analysis is supported via conditionals/sensitivities over trace variables.

To explore these themes, we study CoT through a language-design lens, asking: *What would a programming language for CoT need in order to represent traces, reason over them, and derive new inference strategies?* We derive Copper, a simple discrete probabilistic programming language with a denotational semantics supporting exact, tractable inference, following the compilation-based approaches of Dries et al. (2015); Fierens et al. (2015); Holtzen et al. (2020); Garg et al. (2024); Moy et al. (2025) *inter alia*. We describe the language and explore how it can be used both to reason formally about the limitations of trace likelihood and to develop test-time inference strategies that more effectively combine direct model signals across reasoning steps in CoT.

2 A simple language for CoT

CoT prompting elicits intermediate natural-language reasoning steps before a final answer. For an input query Q, we write a **CoT trace** as $\mathbf{T} = (T_1, \ldots, T_m)$, where each T_i is an intermediate thought sampled from a language model, $T_i \sim \mathbb{P}_{\mathcal{M}}(\cdot \mid \mathbf{T}_{<i}, Q)$, and the final answer A is sampled from $A \sim \mathbb{P}_{\mathcal{M}}(\cdot \mid \mathbf{T}, Q)$. A common probabilistic view of CoT is the **latent-trace model** (Dohan et al., 2022; Xiang et al., 2025), where answer probabilities marginalize over possible reasoning traces: $\mathbb{P}_{\text{CoT}}(A \mid Q) = \sum_{\mathbf{T}} \mathbb{P}_{\mathcal{M}}(A, \mathbf{T} \mid Q)$. Our goal is to make the internal structure of these traces explicit: steps may be uncertain, later steps and answers may depend on earlier

[*]Equal contribution.

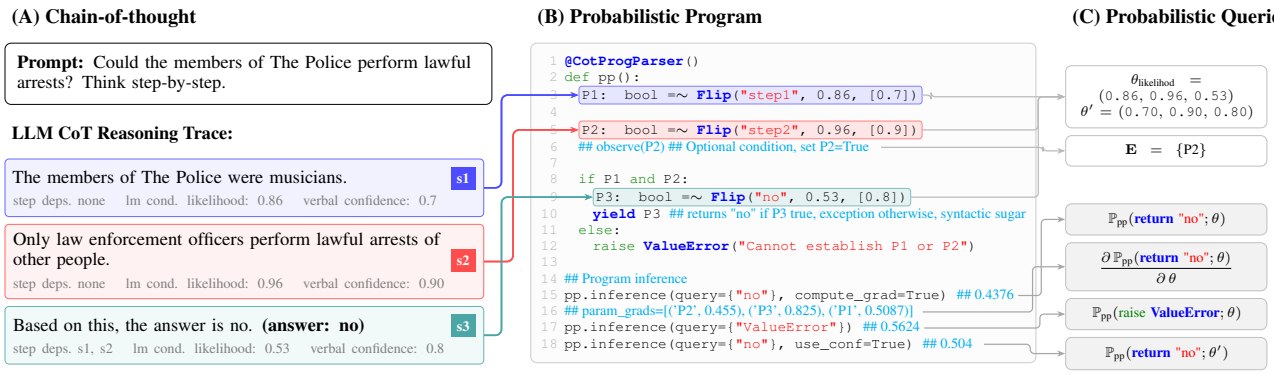


Figure 1: **Copper models CoT traces as probabilistic programs.** (**A**) An example query and reasoning trace paired with a direct translation into a probabilistic program in Copper (**B**). Reasoning steps become stochastic choices (parameterized by some θ such as **lm step likelihood** $\theta_{\text{likelihood}}$), dependencies control flow, and answers/failures return values. The program supports exact probabilistic queries **(C)** over traces, including answer marginals (**lines 15–18**), conditioning via **`observe(.)`** (**line 6**), alternative weight parameterizations θ' (**line 18**), and sensitivity analysis (**line 16**).

ones. We do this by defining a simple, Python-like language for traces $\mathbf{T}$, which we introduce next.

Syntax The syntax of our language is summarized in Figure 2a with the intermediate reasoning steps of CoT being a first class member in Copper. Each thought T_i in $\mathbf{T}$ is represented by a Boolean random variable P, sampled with probability <p> using the probabilistic assignment **`Flip(.)`**. Step dependencies are represented by ordinary *if-then* control flow over Boolean conditions <bool condition> defined over one or more step variables. Reasoning errors are represented as exceptions using **`ValueError`**. Background constraints are encoded using **`observe(.)`**, and answers are return values. Mutually exclusive alternatives are represented using the **`Categorical(.)`** construct, and new variables can be defined via **`Factor(.)`** (see below).

Semantics Probabilistic programs define distributions over executions σ. In our language, an execution assigns values to stochastic decision variables (e.g., the step variables P1, P2, P3 in Figure 1), which determine which branches are taken and which program events are reached. Probabilistic queries therefore reduce to **reachability**: computing the probability of reaching a return value, exception, or intermediate point. We encode reachability using **guards** (Dijkstra, 1975): Boolean formulas over stochastic step variables that characterize when deterministic program events occur (see Figure 2b). Let $\mathcal{V}_{\text{dec}}$ be the stochastic decision variables, $\mathcal{V}_{\neg\text{dec}}$ be the deterministic program-event variables (e.g., returns, exceptions), and define an execution function $\sigma : \mathcal{V}_{\text{dec}} \to \{\text{true}, \text{false}\}$. Each $v \in \mathcal{V}_{\neg\text{dec}}$ is true exactly when at least one of its reachability guards (referred to as $guards(v)$) is satisfied. We formally define Copper and its compilation to weighted Boolean formulae in Section C.

Formally, we define the **program formula** F and **evidence formula** E in Figure 2c. A query q is any Boolean condition over program variables, such as $q =$ `return "no"` and $q =$ **`ValueError`**. Its probability, conditioned on optional evidence, is computed in Figure 2d by **weighted model counting** (WMC) (Chavira & Darwiche, 2008; Fierens et al., 2015): $\text{WMC}(\phi;\theta) := \sum_{\sigma\in\text{Exec}(\phi)} \prod_{\mathsf{P}\in\mathcal{V}_{\text{dec}}} w_{\mathsf{P}}(\sigma(\mathsf{P}))$ where $\text{Exec}(\phi)$ is the set of Boolean executions satisfying ϕ, and $w_{\mathsf{P}}(\text{true}) = \theta_{\mathsf{P}}, w_{\mathsf{P}}(\text{false}) = 1 - \theta_{\mathsf{P}}$ for P =~ **`Flip(·, θP)`**. Our implementation follows this compilation-based approach (Holtzen et al., 2020), translating programs into tractable circuits (Darwiche, 2011, 2021) that permit exact WMC.

Table 1: Program schemata for modeling CoT likelihood and the latent trace model.

$k-$**answer program** F_a

```
## Answer 1
P_{1,1}   =~ Flip(s_{1,1}, θ_{1,1}) ...
P_{1,n_1} =~ Flip(s_{1,n_1}, θ_{1,n_1}) ...
## Answer k
P_{k,1}   =~ Flip(s_{k,1}, θ_{k,1}) ...
P_{k,n_k} =~ Flip(s_{k,n_k}, θ_{k,n_k})
if P_{1,1} and ... and P_{1,n_1}:
    return a_1 ...
elif P_{k,1} and ... and P_{k,n_k}:
    return a_k
else:
    raise ValueError('Error')
```

$k-$**chain program** F_c

```
## Trace 1
P_{1,1}   =~ Flip(s_{1,1}, θ_{1,1}) ...
P_{1,n_1} =~ Flip(s_{1,n_1}, θ_{1,n_1}) ...
## Trace k
P_{k,1}   =~ Flip(s_{k,1}, θ_{k,1}) ...
P_{k,n_k} =~ Flip(s_{k,n_k}, θ_{k,n_k})
A_1, .. A_k, ¬A =~ Categorical(
prod(θ_{1,:}), ... , prod(θ_{k,:})
1 - (prod(θ_{1,:}) + .. + prod(θ_{k,:}))
)
if A_1 or ... or A_k:
    return a
```

3 What can we do with this simple language?

Formal reasoning about CoT Although Copper is minimal— effectively an IMP-style language (Winskel, 1993) extended with probabilistic primitives—it already recovers familiar CoT quantities as ordinary program queries. Table 1 gives two programs: the first models CoT likelihood while the second models the latent-trace model for a single answer. We state the first correspondence below.

Proposition 1 (*CoT likelihood*). *Consider the case of the k-answer program F_a with $k = 1$. Let* $\mathbf{T}^{a_1} =$

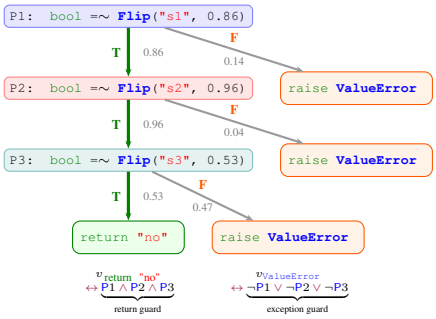
(a) **Core syntax of Copper and grounding in CoT**

CoT Concept	Syntax	Program Vars
Steps are probabilistic Booleans	`P =~ Flip(<str>, <p>)`	$\mathcal{V}_{\text{dec}}$
Step dependencies are controls	`if <bool condition>:` `  <new fact or answer branch v>`	guards(v)

Reasoning Errors are Exceptions	`if <bool condition>:`
	`  <new fact or answer branch v>`
	`else:`
	`  raise ValueError(<error>)`
		$\mathcal{V}_{\neg\text{dec}}$

Answers are returns	`return <answer output>`	$\mathcal{V}_{\neg\text{dec}}$
Conditions are observes	`observe(<bool condition e>)`	evidence
Chains/answers can be unique	$\texttt{A}_1,..,\texttt{A}_n$ `=~ Categorical(<p1, ..>)`	$\mathcal{V}_{\text{dec}}$
Step scores can be aggregated	`P =~ Factor(<str>,` $f_m(p_1,\ldots,p_m)$`)`	$\mathcal{V}_{\text{dec}}$

(b) **Reachability semantics**



(c) **Boolean program encoding**

$$\mathsf{F} := \bigwedge_{v \in \mathcal{V}_{\neg\text{dec}}} \left(v \leftrightarrow \bigvee_{g \in \text{guards}(v)} g \right) \qquad \mathsf{E} := \bigwedge_{e \in \text{evidence}} e$$

(d) **Probabilistic query computation**

$$\mathbb{P}_{\mathsf{F}}(q \mid \mathsf{E}; \theta) = \frac{\text{WMC}(\mathsf{F} \wedge \mathsf{E} \wedge q;\, \theta)}{\text{WMC}(\mathsf{F} \wedge \mathsf{E};\, \theta)}$$

Figure 2: **Syntax and semantics of Copper. (a)** Core language constructs. **(b) Reachability semantics**: each path corresponds to a program execution, with path weights determined by local decision probabilities; guards specify the Boolean conditions under which evidence-conditioned program points are reached. **(c)** Boolean encoding of the program and evidence formulas. **(d)** Exact conditional query probabilities computed by weighted model counting (WMC).

$(s_{1,1}, \ldots, s_{1,n_1})$ *be a CoT trace for a query* Q *whose final step returns answer* a_1*. If each flip probability is set to the* ***LM conditional step likelihood*** $\theta_{1,j} = \mathbb{P}_{\mathcal{M}}(s_{1,j} \mid s_{1,<j}, Q)$*, then* $\mathbb{P}_{\mathsf{F}_a}(\neg\texttt{ValueError}; \theta) = \mathbb{P}_{\mathcal{M}}(\mathbf{T}^{a_1} \mid Q)$.

In other words, under this **lm likelihood** parameterization, the probability of not reaching an exception corresponds exactly to the likelihood of a single CoT trace; its negative logarithm therefore yields the negative log-likelihood loss, showing the close connection between program inference and learning (Xu et al., 2018). In App. B, we prove an analogous correspondence between F_c and the latent-trace marginal and discuss the semantics of $k > 1$ and branching.

These correspondences are elementary, but their role is foundational: *standard CoT likelihoods and latent-trace marginals arise directly from the same reachability semantics used for arbitrary program queries*. The programming formulation makes assumptions about traces explicit and formally analyzable, while the schemata provide declarative templates for constructing richer reasoning strategies.

Understanding the limitations of likelihood Advanced test-time reasoning methods often rely on search, external solvers, or reward models (Yao et al., 2023, 2022; Gao et al., 2023; Zhong et al., 2026), typically at the cost of additional model calls. We instead ask whether richer reasoning can be obtained from fewer calls by structuring and using the model's own signals more effectively. The probabilistic-program view helps both to formulate such strategies and to expose the limitations of likelihood as a sole scoring signal.

As an example, **multi-answer CoT** is a novel prompt strategy below that **instructs** the model to produce multiple candidate reasoning paths and answers directly in a single call. The resulting trace translates directly into a branching program, which obviates the need for many forward calls.

Instruction: Please answer each question thinking step-by-step. Rather than giving a single answer, however, I want you to express different possible reasoning paths. Number your steps and identify step dependencies.
Query: Would a sea kayak fit into a normal car?
Example Trace *(Llama-3.3-70B-turbo-instruct)* **1.** A sea kayak is typically around 14-18 feet long **2.** Most cars have a maximum interior length of around 6-8 feet **3.** Based on 1 and 2, the answer is: no (ref. 1, 2) (final answer: no) **4.** Some cars have roof racks that can carry long items like kayaks **5.** Alternatively, based on 4, the answer is: yes (ref. 4) (final answer: yes)
Step likelihoods $\theta_{\text{likelihood}}$ (length normalized) (*for no*) P1 0.839 P2 0.99 P3 0.951. (*for yes*) P4 0.921 P5 0.92
Program sketch: `if P1 and P2 and P3: return "no" elif P4 and P5: return "yes"`

This representation permits candidate paths to share, contrast, or revise steps explicitly, without a separate process reward model. In the example, however, exact inference assigns greater probability to *yes*, even though the *no* path has slightly greater average step support. The reason is structural: conjunction multiplies step probabilities, causing raw trace likelihood to penalize longer chains.

Proposition 2 (*Length confound*)**.** *Let* t_1, t_2 *be two CoT chains with lengths* n_1, n_2*, log-likelihoods* $\ell_1, \ell_2 < 0$*, and mean log-likelihoods* $\bar{\ell}_i = \ell_i / n_i$*. If* $\bar{\ell}_1 > \bar{\ell}_2$*, then likelihood prefers* t_1 *over* t_2 *only under the following condition:* $\ell_1 > \ell_2 \iff \frac{n_1}{n_2} < \frac{|\bar{\ell}_2|}{|\bar{\ell}_1|}$.

Thus, even when t_1 has greater mean log-likelihood, it is preferred only if $n_1 < n_2 \frac{|\bar{\ell}_2|}{|\bar{\ell}_1|}$. Any inference procedure that

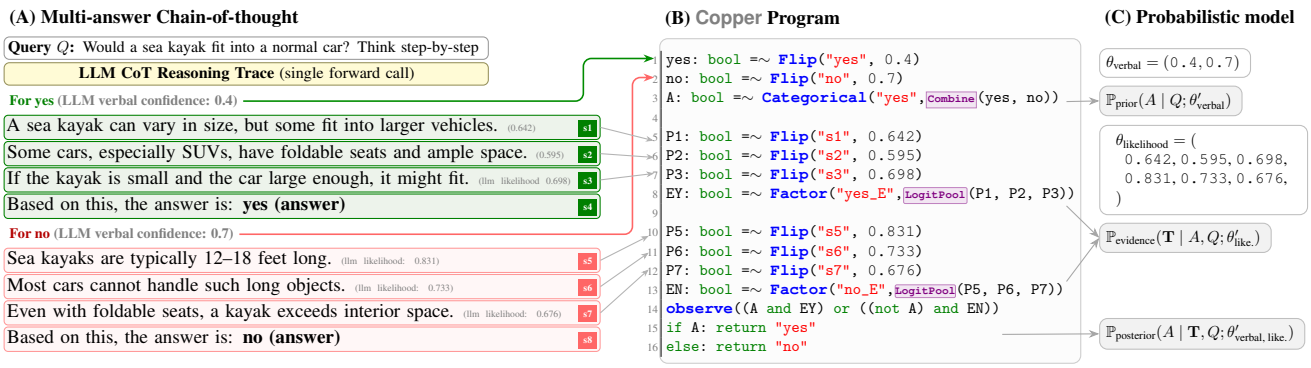


Figure 3: **Bayesian multi-answer CoT**, a prompting strategy derived from our framework that elicits multiple candidate answers and supporting reasoning paths. **(A)** An example `gpt-4o` trace produced in a single forward call. **(B)** The corresponding Copper program and its underlying probabilistic model **(C)**, which uses differentiable operators to aggregate step-level evidence using a mixture of LM likelihood signals ($\theta_{\text{likelihood}}$) and verbal probabilistic feedback (θ_{verbal}).

ranks chains by raw likelihood therefore inherits a length confound. This elementary result illustrates a broader benefit of the programming formulation: *candidate inference strategies can be represented declaratively and their structural biases analyzed formally before deployment.*

Adding differentiable operators Returning to our language-design question, this failure shows that CoT programs must support more than conjunction and likelihood multiplication. We therefore introduce **`Factor`**, which assigns program weights through a differentiable aggregation $f_m(p_1, \ldots, p_m)$ of earlier scores, enabling normalization, confidence aggregation, feedback, and other interactions. This lifts the language from probabilistic modeling of CoT to differentiable programming over LLM outputs (Lew et al., 2023a), where richer computation graphs for CoT can be constructed and optimized.

4 Ongoing Case Studies

We implement Copper as an embedded DSL in Python, following Saad et al. (2021). Programs compile to SDDs through `PySDD` (KULeuven, 2026) for exact WMC, with `PyTorch` (Paszke et al., 2019) providing AutoDiff support. We briefly describe how Copper supports novel robustness analysis and the design of new inference strategies.

Measuring robustness The program representation enables a notion of robustness inspired by formal verification (Kwiatkowska et al., 2007): *how much must the semantics θ of a reasoning trace change before its output flips?* We define a robustness margin over step probabilities, answer priors, and other program parameters. Exact differentiable inference provides the score gap and its gradient, yielding a local estimate of the smallest perturbation needed to change the prediction. The same analysis can be applied across alternative program structures and across feedback from multiple LLMs, revealing disagreement between models.

This analysis also gives a criterion for designing better reasoning procedures: *inference strategies should not only select accurate answers, but produce decisions that remain stable under small changes to their supporting evidence.*

Deriving robust prompting strategies Motivated by this analysis, we investigate **Bayesian multi-answer CoT**, in which a single call elicits multiple candidate answers. Rather than sampling one trace per call, the model produces multiple candidate answers, answer-conditioned reasoning paths, step dependencies, and verbal confidence estimates in a single response. The output is translated into the probabilistic program shown in Figure 3.

The strategy uses the Bayesian factorization $\mathbb{P}_{\text{post}}(A \mid \mathbf{T}, Q) \propto \mathbb{P}_{\text{evidence}}(\mathbf{T} \mid A, Q)\mathbb{P}_{\text{prior}}(A \mid Q)$. We parameterize these terms by reversing the usual generation order: the LLM first produces candidate answers and initial confidence estimates, then generates reasoning conditioned on each answer. Answer confidences define the priors, while step likelihoods, verbal confidence, and dependencies define the evidence terms through differentiable aggregation (e.g., LogitPool combines step-level signals by averaging their log-odds). In ongoing experiments, these Bayesian programs exhibit larger robustness margins than likelihood-only formulations, suggesting that the same structural choices that improve inference also yield less brittle decisions.

5 Conclusion

We introduced Copper, a language for representing CoT traces and reasoning over them with exact, differentiable inference. We show how such programmatic representations make assumptions about LLM reasoning explicit and amenable to formal analysis. We see Copper as a first step towards a broader language-design agenda for modeling complex LLM reasoning while retaining formal and tractable analysis.

References

Ahmed, K., Teso, S., Chang, K.-W., Van den Broeck, G., and Vergari, A. Semantic probabilistic layers for neuro-symbolic learning. *Advances in Neural Information Processing Systems*, 35:29944–29959, 2022.

Ahmed, K., Chang, K.-W., and Broeck, G. V. d. Controllable generation via locally constrained resampling. *arXiv preprint arXiv:2410.13111*, 2024.

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

Bi, Z., Han, K., Liu, C., Tang, Y., and Wang, Y. Forest-of-thought: Scaling test-time compute for enhancing llm reasoning. *arXiv preprint arXiv:2412.09078*, 2024.

Blondel, M. and Roulet, V. The elements of differentiable programming. *arXiv preprint arXiv:2403.14606*, 2024.

Calanzone, D., Teso, S., and Vergari, A. Logically consistent language models via neuro-symbolic integration. *arXiv preprint arXiv:2409.13724*, 2024.

Chavira, M. and Darwiche, A. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172 (6-7):772–799, 2008.

Chen, Q., Qin, L., Liu, J., Peng, D., Guan, J., Wang, P., Hu, M., Zhou, Y., Gao, T., and Che, W. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models. *arXiv preprint arXiv:2503.09567*, 2025.

Cheng, J., Richardson, K., and Chin, P. Analytica: Soft propositional reasoning for robust and scalable llm-driven analysis. *Proceedings of ICLR*, 2026.

Clark, K. L. Negation as failure. In *Logic and data bases*, pp. 293–322. Springer, 1977.

Cozman, F. G. and Mauá, D. D. The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference. *International Journal of Approximate Reasoning*, 125:218–239, 2020.

Darwiche, A. Sdd: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, 2011.

Darwiche, A. Tractable boolean and arithmetic circuits. In *Neuro-Symbolic Artificial Intelligence: The State of the Art*, pp. 146–172. IOS Press, 2021.

Darwiche, A. and Marquis, P. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

De Raedt, L. and Kimmig, A. Probabilistic (logic) programming concepts. *Machine Learning*, 100:5–47, 2015.

De Raedt, L., Kimmig, A., and Toivonen, H. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of IJCAI*, pp. 2462–2467, 2007.

De Raedt, L., Kersting, K., Natarajan, S., and Poole, D. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis lectures on artificial intelligence and machine learning*, 10(2):1–189, 2016.

Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

Dohan, D., Xu, W., Lewkowycz, A., Austin, J., Bieber, D., Lopes, R. G., Wu, Y., Michalewski, H., Saurous, R. A., Sohl-Dickstein, J., et al. Language model cascades. *arXiv preprint arXiv:2207.10342*, 2022.

Dries, A., Kimmig, A., Meert, W., Renkens, J., Van den Broeck, G., Vlasselaer, J., and De Raedt, L. Problog2: Probabilistic logic programming. In *Joint european conference on machine learning and knowledge discovery in databases*, pp. 312–315. Springer, 2015.

Feng, Y., Zhou, B., Lin, W., and Roth, D. Bird: A trustworthy bayesian inference framework for large language models. *Proceedings of ICLR*, 2025.

Feng, Y., Weir, N., Bostrom, K., Bayless, S., Cassel, D., Chaudhary, S., Kiesl-Reiter, B., and Rangwala, H. Vericot: Neuro-symbolic chain-of-thought validation via logical consistency checks. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=zHuV3Vatov.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Garg, P., Holtzen, S., Van den Broeck, G., and Millstein, T. Bit blasting probabilistic programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):865–888, 2024. ISSN 2475-1421. doi: 10.1145/3656412. URL http://dx.doi.org/10.1145/3656412.

Garg, P., Geh, R. L., Israel, D., Millstein, T., Richardson, K., and Broeck, G. V. d. Probabilistic programs of thought. *arXiv preprint arXiv:2604.17290*, 2026.

Gordon, A. D., Henzinger, T. A., Nori, A. V., and Rajamani, S. K. Probabilistic programming. In *Future of software engineering proceedings*, pp. 167–181. 2014.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Holtzen, S., Millstein, T., and den Broeck, G. V. Symbolic exact inference for discrete probabilistic programs, 2019. URL https://arxiv.org/abs/1904.02079.

Holtzen, S., Van den Broeck, G., and Millstein, T. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–31, 2020.

Huang, Y., Chen, Y., Zhang, H., Li, K., Zhou, H., Fang, M., Yang, L., Li, X., Shang, L., Xu, S., et al. Deep research agents: A systematic examination and roadmap. *arXiv preprint arXiv:2506.18096*, 2025.

Katoen, J.-P., Gretz, F., Jansen, N., Kaminski, B. L., and Olmedo, F. Understanding probabilistic programs. In *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015, Proceedings*, pp. 15–32. Springer, 2015.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., and Sabharwal, A. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

KULeuven. Pysdd, 2026. URL https://github.com/ML-KULeuven/PySDD.

Kwiatkowska, M., Norman, G., and Parker, D. Stochastic model checking. In *Formal Methods for Performance Evaluation*, pp. 220–270. Springer, 2007.

Lanham, T., Chen, A., Radhakrishnan, A., Steiner, B., Denison, C., Hernandez, D., Li, D., Durmus, E., Hubinger, E., Kernion, J., et al. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*, 2023.

Lew, A. K., Huot, M., Staton, S., and Mansinghka, V. K. Adev: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):121–153, January 2023a. ISSN 2475-1421. doi: 10.1145/3571198. URL http://dx.doi.org/10.1145/3571198.

Lew, A. K., Zhi-Xuan, T., Grand, G., and Mansinghka, V. K. Sequential monte carlo steering of large language models using probabilistic programs. *arXiv preprint arXiv:2306.03081*, 2023b.

Li, Z., Huang, J., and Naik, M. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

Liu, A., Liu, X., Zhao, D., Niepert, M., Liang, Y., and Broeck, G. V. d. Tractable transformers for flexible conditional generation. *arXiv preprint arXiv:2502.07616*, 2025.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., and De Raedt, L. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

Mo, S. and Xin, M. Tree of uncertain thoughts reasoning for large language models. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 12742–12746. IEEE, 2024.

Moy, C., Czenszak, J., Li, J. M., Marshall, B., and Holtzen, S. Roulette: A language for expressive, exact, and efficient discrete probabilistic programming. *Proceedings of the ACM on Programming Languages*, 9(PLDI):2081–2105, 2025.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pearl, J. *Probabilistic reasoning in intelligent systems*, volume 88. Elsevier, 1988.

Pfeffer, A. Practical probabilistic programming. In *International Conference on Inductive Logic Programming*, pp. 2–3. Springer, 2010.

Quan, X., Valentino, M., Dennis, L. A., and Freitas, A. Verification and refinement of natural language explanations through LLM-symbolic theorem proving. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 2933–2958, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.172. URL https://aclanthology.org/2024.emnlp-main.172/.

Richardson, K., Srikumar, V., and Sabharwal, A. Understanding the logic of direct preference alignment through logic. *Proceedings of ICML*, 2025.

Saad, F. A., Rinard, M. C., and Mansinghka, V. K. Sppl: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 804–819, 2021.

Stechly, K., Valmeekam, K., and Kambhampati, S. Chain of thoughtlessness? an analysis of cot in planning. *Advances in Neural Information Processing Systems*, 37:29106–29141, 2024.

Turpin, M., Michael, J., Perez, E., and Bowman, S. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36: 74952–74965, 2023.

Van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, pp. 431–445. Springer, 2004.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

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

Wu, M., Zhu, T., Han, H., Zhang, X., Shao, W., and Chen, W. Chain-of-tools: Utilizing massive unseen tools in the cot reasoning of frozen language models. *arXiv preprint arXiv:2503.16779*, 2025.

Xiang, V., Snell, C., Gandhi, K., Albalak, A., Singh, A., Blagden, C., Phung, D., Rafailov, R., Lile, N., Mahan, D., et al. Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought. *arXiv preprint arXiv:2501.04682*, 2025.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Broeck, G. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. In *Proceedings of ICML*, pp. 5498–5507, 2018.

Yang, L., Yu, Z., Zhang, T., Cao, S., Xu, M., Zhang, W., Gonzalez, J. E., and Cui, B. Buffer of thoughts: Thought-augmented reasoning with large language models. *Advances in Neural Information Processing Systems*, 37: 113519–113544, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Ye, X., Chen, Q., Dillig, I., and Durrett, G. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36:45548–45580, 2023.

Yeo, E., Tong, Y., Niu, M., Neubig, G., and Yue, X. Demystifying long chain-of-thought reasoning in llms. *arXiv preprint arXiv:2502.03373*, 2025.

Yidou-Weng, G., Wang, B., and Van den Broeck, G. Trace back from the future: A probabilistic reasoning approach to controllable language generation. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

Zhang, H., Dang, M., Peng, N., and Van den Broeck, G. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pp. 40932–40945. PMLR, 2023.

Zhong, W., Liu, C., Wu, Y., Tan, B., Sun, C., Wang, Y., Liu, X., and Kuang, K. P2s: Probabilistic process supervision for general-domain reasoning question answering. *arXiv preprint arXiv:2601.20649*, 2026.

A Extended Related Work

CoT and Variants Our work takes inspiration from the literature on CoT and its variants that couple CoT with advanced search techniques (Yao et al., 2023; Mo & Xin, 2024; Besta et al., 2024; Bi et al., 2024; Yang et al., 2024), external tools (Ye et al., 2023; Gao et al., 2023; Wu et al., 2025) and other forms of test-time scaling (Khot et al., 2022; Wang et al., 2022; Feng et al., 2025; Cheng et al., 2026). In contrast, our work focuses not on a specific inference strategy but on a general formalism for CoT. As such, our work is closest to

Dohan et al. (2022), who propose a related graphical model view of CoT; however, their work is primarily conceptual, whereas we define an explicit language for CoT with a formal semantics and use it to derive new formal and empirical results. We also follow other work such as Beurer-Kellner et al. (2023); Quan et al. (2024); Feng et al. (2026) that attempts to ground LLM inference in programming.

Probabilistic and Differential Programming Our technical framework draws on probabilistic programming (Pfeffer, 2010; Gordon et al., 2014; Katoen et al., 2015; Van de Meent et al., 2018), statistical relational learning (De Raedt et al., 2016), and differentiable programming (Blondel & Roulet, 2024). We build our approach and language most directly on discrete probabilistic programming (Holtzen et al., 2020; Saad et al., 2021), where programs define distributions over finite executions via random choices, branching, conditioning, and return values. Exact inference in this setting can be performed by reducing probabilistic queries to weighted model counting over compiled logical representations (Chavira & Darwiche, 2008; Darwiche & Marquis, 2002). As such, our focus on exact inference differs from other approaches that mix probabilistic programming with approximate inference and LLMs such as Wong et al. (2023); Lew et al. (2023b).

While we use imperative-style program representations, our work work takes inspiration from probabilistic logic programming and other declarative-style approaches to probabilistic programming (De Raedt et al., 2007; Dries et al., 2015; Manhaeve et al., 2018; Li et al., 2023; Cozman & Mauá, 2020). Indeed, we note that the Boolean nature of our programs and our program encoding in Figure 2c operationalizes a Clark-style completion semantics (Clark, 1977), however our language differs in having more complex differentiable operators.

Tractable Probabilistic Modeling for LLMs Finally, our work takes inspiration from work on using tractable probabilistic models for LLM modeling problems (Ahmed et al., 2024; Zhang et al., 2023; Liu et al., 2025; Yidou-Weng et al., 2025; Garg et al., 2026), and other approaches involving tractable neuro-symbolic inference (Xu et al., 2018; Ahmed et al., 2022; Richardson et al., 2025; Calanzone et al., 2024). In contrast to this work, much of which centers around constrained decoding problems, we focus on applications of probabilistic modeling of CoT.

B Formal Results

B.1 The $k-$answer program for $k = 1$

Proposition 1 (*CoT likelihood*). *Consider the case of the k-answer program* F_a *with* $k = 1$. *Let* $\mathbf{T}^{a_1} = (s_{1,1}, \ldots, s_{1,n_1})$ *be a CoT trace for a query* Q *whose final step returns answer* a_1. *If each flip probability is set to the* ***LM conditional step likelihood*** $\theta_{1,j} = \mathbb{P}_{\mathcal{M}}(s_{1,j} \mid s_{1,<j}, Q)$, *then* $\mathbb{P}_{\mathsf{F}_a}(\neg\texttt{ValueError}; \theta) = \mathbb{P}_{\mathcal{M}}(\mathbf{T}^{a_1} \mid Q)$.

Proof. Let the following formula be the guard formula for reaching answer a_1:

$$G_1 = \bigwedge_{j=1}^{n_1} \mathsf{P}_{1,j}$$

which is satisfied exactly when **`ValueError`** is not encountered. Hence, under the semantics of the **`Flip`**:

$$\mathbb{P}_{\mathsf{F}_a}(\neg\texttt{ValueError}; \theta) = \mathbb{P}_{\mathsf{F}_a}(G_1; \theta) = \prod_{j=1}^{n_1} \theta_{1,j}.$$

Substituting the assumed model conditionals and applying the chain rule then yields

$$\begin{aligned}\prod_{j=1}^{n_1} \theta_{1,j} &= \prod_{j=1}^{n_1} \mathbb{P}_{\mathcal{M}}(s_{1,j} \mid s_{1,<j}, Q) \\ &= \mathbb{P}_{\mathcal{M}}(s_{1,1}, \ldots, s_{1,n_1} \mid Q) \\ &= \mathbb{P}_{\mathcal{M}}(\mathbf{T}^{a_1} \mid Q).\end{aligned}$$

which establishes the correspondence. □

B.2 The $k-$chain program

Below we state an analogous result for the $k-$chain program in Table 1. In this program, we use **`prod`**$(\theta_{j:})$ as a helper to denote the product of the step probabilities for each jth CoT. Importantly, we note the presence of a negated variable $\neg A$ in **`Categorical`**, which denotes the negation of all choices and follows the semantics of annotated disjunctions in probabilistic logic programming (Vennekens et al., 2004).

Proposition 3 (*Finite latent-trace approximation*). *For the k-chain program* F_c *(Table 1), suppose each disjunct* i *encodes a distinct CoT trace* $\mathbf{T}_i^a = (s_{i,1}, \ldots, s_{i,n_i})$ *for a query* Q *whose final step returns answer* a. *If each flip probability is set to* $\theta_{i,j} = \mathbb{P}_{\mathcal{M}}(s_{i,j} \mid s_{i,<j}, Q)$ *then*

$$\mathbb{P}_{\mathsf{F}_c}(\texttt{return a}; \theta) = \mathbb{P}_{\text{CoT}_k}(A = a \mid Q)$$

with $\mathbb{P}_{\text{CoT}_k}(A = a \mid Q) := \sum_{i=1}^{k} \mathbb{P}_{\mathcal{M}}(A = a, \mathbf{T} = \mathbf{T}_i^a \mid Q)$ *being a finite approximation of the latent-trace marginal.*

Proof. Let G_i be the success guard for the i-th disjunct in F_c: Since the k-chain program returns a whenever any encoded chain succeeds, the return guard is

$$\texttt{return a} \iff \bigvee_{i=1}^{k} G_i.$$

By the assumed weighting and the chain rule,

$$\begin{aligned}\mathbb{P}_{\mathsf{F}_c}(G_i; \theta) &= \prod_{j=1}^{n_i} \mathbb{P}_{\mathcal{M}}(s_{i,j} \mid s_{i,<j}, Q) \\ &= \mathbb{P}_{\mathcal{M}}(A = a, \mathbf{T} = \mathbf{T}_i^a \mid Q),\end{aligned}$$

where the final equality uses that $\mathbf{T}_i^a$ is a trace supporting answer a, with the answer included in the encoded trace as specified in the proposition. Since the trace identities $\mathbf{T} = \mathbf{T}_i^a$ are distinct in virtue of being `Categorical`, the corresponding trace events are mutually exclusive in the latent-trace model. Therefore,

$$\mathbb{P}_{\mathsf{F}_c}(\texttt{return a}) = \mathbb{P}_{\mathsf{F}_c}\left(\bigvee_{i=1}^{k} G_i; \theta\right)$$
$$= \sum_{i=1}^{k} \mathbb{P}_{\mathsf{F}_c}(G_i; \theta) = \sum_{i=1}^{k} \mathbb{P}_{\mathcal{M}}(A = a, \mathbf{T} = \mathbf{T}_i^a \mid Q).$$

which establishes the result. □

B.3 The $k-$answer program for $k > 1$

One subtle detail in the above program and proof is the use of a categorical choice, which imposes a uniqueness constraint that allows only one of the k chains to be selected. Removing this constraint and assigning each branch a disjoint set of independent decision variables yields a different semantics. To see this in relation to the $k-$answer program, define each guard G_i as above and its corresponding likelihood L_i as

$$G_i = \bigwedge_{j=1}^{n_i} \mathsf{P}_{i,j}, \qquad L_i = \mathbb{P}_{\mathsf{F}_a}(G_i; \theta) = \prod_{j=1}^{n_i} \theta_{i,j}.$$

The probability that at least one independent branch succeeds is then

$$\mathbb{P}_{\mathsf{F}_a}\left(\bigvee_{i=1}^{k} G_i; \theta\right) = 1 - \prod_{i=1}^{k}(1 - L_i)$$

which corresponds to a Noisy-OR (Pearl, 1988) or probabilistic disjunction under independence. Based on this, we generalize Prop. 1.

Proposition 4 (*Noisy-or*). *For the k-answer program F_a with $k > 1$, let each $\mathbf{T}^{a_i} = (s_{i,1}, \ldots, s_{i,n_i})$ be a CoT trace for a query Q whose final step returns answer a_i. If each flip probability is set to $\theta_{i,j} = \mathbb{P}_{\mathcal{M}}(s_{i,j} \mid s_{i,<j}, Q)$, then*

$$\mathbb{P}_{\mathsf{F}_a}(\neg\texttt{ValueError}; \theta) = 1 - \prod_{i}^{k} 1 - P_{\mathcal{M}}(\mathbf{T}^{a_i} \mid Q)$$

or the probability of the language model emiting one or more of the of the k CoT traces.

Proof. The program avoids `ValueError` exactly when at least one guard G_i succeeds. Because the guards use disjoint decision variables, they are independent. Applying the chain rule we get:

$$\mathbb{P}_{\mathsf{F}_a}(G_i; \theta) = \prod_{j=1}^{n_i} \theta_{i,j} = \mathbb{P}_{\mathcal{M}}(\mathbf{T}^{a_i} \mid Q).$$

then applying the probability of a union of independent events yields the target result. □

By adding the categorical guards as in the previous program, we note that resulting program instead computes in the total probability mass that a language assigns to the k CoTs. Taken together, these results highlight three related semantics. A single branch recovers ordinary CoT likelihood; multiple independent branches produce a noisy-OR over independent trace attempts; and replacing the independent guards with a categorical trace choice recovers the total probability mass of mutually exclusive traces.

B.4 The length confound

Proposition 2 (*Length confound*). *Let t_1, t_2 be two CoT chains with lengths n_1, n_2, log-likelihoods $\ell_1, \ell_2 < 0$, and mean log-likelihoods $\bar{\ell}_i = \ell_i / n_i$. If $\bar{\ell}_1 > \bar{\ell}_2$, then likelihood prefers t_1 over t_2 only under the following condition: $\ell_1 > \ell_2 \iff \frac{n_1}{n_2} < \frac{|\bar{\ell}_2|}{|\bar{\ell}_1|}$.*

Proof. Raw likelihood prefers t_1 iff $\ell_1 > \ell_2$. Since $\bar{\ell}_i = \ell_i / n_i$, we have $\ell_i = n_i \bar{\ell}_i$, so this is equivalent to

$$n_1 \bar{\ell}_1 > n_2 \bar{\ell}_2.$$

Because log-likelihoods are negative, write $\bar{\ell}_i = -|\bar{\ell}_i|$. Then $-n_1|\bar{\ell}_1| > -n_2|\bar{\ell}_2|$. Multiplying by -1 then reverses the inequality: $n_1|\bar{\ell}_1| < n_2|\bar{\ell}_2|$. Dividing both sides by $n_2|\bar{\ell}_1| > 0$ gives

$$\frac{n_1}{n_2} < \frac{|\bar{\ell}_2|}{|\bar{\ell}_1|}.$$

Therefore

$$\ell_1 > \ell_2 \quad \iff \quad \frac{n_1}{n_2} < \frac{|\bar{\ell}_2|}{|\bar{\ell}_1|}.$$

□

C Syntax of COPPER and Compilation to Weighted Boolean Formula

Figure 4 provides the formal syntax of Copper and Figure 5 provides the formal compilation rules to weighted Boolean formulae. To perform inference over the chain-of-thought programs represented by Copper, given a set of defined variables, we compile it to a weighted Boolean formula. Formally, the compilation judgement for Copper programs have the form $\Gamma, \Sigma \vdash p \rightsquigarrow (w, g, e)$. Here, Γ refers to a set of variables, Σ refers to a mapping from strings to variables, weights w refers to a mapping from variables to probabilities, guards g refers to a mapping from tuples of string and deterministic variables to Boolean formulae, evidence e refers to a Boolean formulae on which the probability distribution is to be conditioned. These compilation rules closely

$$
\begin{array}{rrll}
\textit{Values} & v & ::= & \texttt{T} \mid \texttt{F} \mid x \\
\textit{Boolean Expressions} & b & ::= & v \mid v \vee b \mid v \wedge b \mid \neg b \\
\textit{Statements} & s & ::= & x =\sim \textbf{Flip}(\theta) \\
 & & \mid & x_1, \ldots x_n =\sim \textbf{Categorical}(\theta_1, \theta_2, \ldots, \theta_n) \\
 & & \mid & x =\sim \textbf{Factor}(f_m(\theta_1, \ldots, \theta_m)) \mid \textbf{observe}(b) \\
 & & \mid & \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p \mid \textbf{return}\ t \mid \textbf{raise}\ t \\
\textit{Program} & p & ::= & s \mid s; p
\end{array}
$$

Figure 4: The syntax of Copper. The metavariable x ranges over variable names, θ ranges over real numbers in the range $[0, 1]$, t ranges over strings and f_m ranges over differentiable functions $[0, 1]^m \to [0, 1]$

follow the rules provided in existing work for compiling discrete PPLs to weighted Boolean formulae (Holtzen et al., 2019). In fact, it is simpler for Copper as it assumes that all variables are assigned distributions only once and all observations are global.

Before we describe the compilation judgement, we first define few helper functions.

- For a mapping w, we define its domain $dom(w) := \{v \mid \exists \theta\ (v, \theta) \in w\}$.
- Given Boolean variables, $x_1, x_2, \ldots x_n$, we define the constraint of exactly one of them being true as $\texttt{exact\_one}(x_1, x_2, \ldots x_n)$

$$
\begin{aligned}
&\texttt{exact\_one}(x_1, x_2, \ldots x_n) \\
&:= \begin{cases} x_1 & n = 1 \\ (x_1 \wedge \bigwedge_{i=2}^{n} \neg x_i) & \\ \vee(\neg x_1 \wedge \texttt{exact\_one}(x_2, \ldots x_n)) & n \neq 1 \end{cases}
\end{aligned}
$$

- If for all variables x occurring in a Boolean formula b, $x \in \Gamma$, then we say that b is well defined in the context Γ, i.e. $\Gamma \vDash b$.
- Given a mapping g from variables to Boolean formulas, and a Boolean formula b, the conjunction $b \wedge g := \{((t, x), f \wedge b) \mid ((t, x), b) \in g\}$.
- Given two mappings g_1 and g_2, their union is defined as

$$
\begin{aligned}
&(g_1 \sqcup g_2)((t, x)) := \\
&\begin{cases} g_1((t, x)) \vee g_2((t, x)) & (t, x) \in dom(g_1) \cap dom(g_2), \\ g_1((t, x)) & (t, x) \in dom(g_1) \setminus dom(g_2), \\ g_2((t, x)) & (t, x) \in dom(g_2) \setminus dom(g_1). \end{cases}
\end{aligned}
$$

- The construct $\texttt{fresh}\ x$ retrieves a fresh variable name x.

Now with the compilation rules, if $\emptyset, \emptyset \vdash p \rightsquigarrow (w, g, e)$, then $\nu_{\neg \text{dec}} = dom(g)$, $guards(v) = g(v)$

$$\Gamma, \Sigma \vdash x =\sim \texttt{Flip}(\theta) \rightsquigarrow (\{x : \theta, \neg x : 1 - \theta\}, \emptyset, \texttt{true})$$

$$\Gamma, \Sigma \vdash x_1, \ldots x_n =\sim \texttt{Categorical}(\theta_1, \ldots \theta_n) \rightsquigarrow (\{x_1 : \theta_1, \ldots, x_n : \theta_n, \neg x_1 : 1, \ldots \neg x_n : 1\}, \emptyset, \texttt{exact\_one}(x_1, \ldots, x_n))$$

$$\Gamma, \Sigma \vdash x =\sim \texttt{Factor}(f_m(\theta_1, \ldots \theta_m)) \rightsquigarrow (\{x : f_m(\theta_1, \ldots \theta_m), \neg x : 1 - f_m(\theta_1, \ldots \theta_m)\}, \emptyset, \texttt{true})$$

$$\frac{\exists x \;\; (t, x) \in \Sigma}{\Gamma, \Sigma \vdash \textbf{return}\ t \rightsquigarrow (\emptyset, \{(t, x) : \texttt{true}\}, \texttt{true})} \qquad \frac{\nexists x \;\; (t, x) \in \Sigma \qquad \texttt{fresh}\;\; x}{\Gamma, \Sigma \vdash \textbf{return}\ t \rightsquigarrow (\emptyset, \{(t, x) : \texttt{true}\}, \texttt{true})}$$

$$\frac{\exists x \;\; (t, x) \in \Sigma}{\Gamma, \Sigma \vdash \textbf{raise}\ t \rightsquigarrow (\emptyset, \{(t, x) : \texttt{true}\}, \texttt{true})} \qquad \frac{\nexists x \;\; (t, x) \in \Sigma \qquad \texttt{fresh}\;\; x}{\Gamma, \Sigma \vdash \textbf{raise}\ t \rightsquigarrow (\emptyset, \{(t, x) : \texttt{true}\}, \texttt{true})}$$

$$\frac{\Gamma, \Sigma \vDash b}{\Gamma, \Sigma \vdash \texttt{observe}(b) \rightsquigarrow (\emptyset, \emptyset, b)}$$

$$\frac{\Gamma \vDash b \qquad \Gamma, \Sigma \vdash p_1 \rightsquigarrow (w_1, g_1, e_1) \qquad \Gamma, \Sigma \cup dom(g_1) \vdash p_2 \rightsquigarrow (w_2, g_2, e_2)}{\Gamma, \Sigma \vdash \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \rightsquigarrow (w_1 \cup w_2, (b \wedge g_1) \sqcup (\neg b \wedge g_2), e_1 \wedge e_2)}$$

$$\frac{\Gamma, \Sigma \vdash s \rightsquigarrow (w_1, g_1, e_1) \qquad \Gamma \cup dom(w_1), \Sigma \cup dom(g_1) \vdash p \rightsquigarrow (w_2, g_2, e_2)}{\Gamma, \Sigma \vdash s; p \rightsquigarrow (w_1 \cup w_2, g_1 \sqcup g_2, e_1 \wedge e_2)}$$

Figure 5: Compilation of Copper programs to weighted Boolean formulas.