Conditional Deontics over Terminals: A Mildly Context-Sensitive Formal Grammar for Constrained Decoding

Anonymous ACL submission

Abstract

With the broad adoption of large language models (LLMs), techniques to guide their generation have become increasingly important, with particularly sophisticated patterns found in constrained decoding frameworks expressed via formal grammars. Recent frameworks such as IterGen expand beyond context-free grammars to achieve some context sensitivity but with drawbacks: expensive backtracking or speculative lookahead, permissive semantics that risk spurious constraint activation, and difficult integration with runtime optimizations like XGrammar. To address these concerns, we propose Conditional Deontics over Terminals (CDoT), a mildly context-sensitive grammar paired with a GPU-accelerated overall O(n^3) parser that can be employed for constrained decoding with LLMs. We show that CDoT can express useful constraints that improve LLM performance in a traditional text-to-SQL task and a complex logic puzzle involving unit propagation.

1 Introduction

As large language models (LLMs) have become more popular, the importance of techniques to guide their generation has also grown. Techniques for steering LLMs target various model states, including activations, attention patterns, MoE router logits, and output logits. These interventions can be applied uniformly, or conditioned on the generation process itself. For example, CAST (Lee et al., 2025) conditions on the cosine similarity between activations and behavior vectors, FASB (Cheng et al., 2025) applies a neural classifier to activations to make steering decisions, and PASTA (Zhang et al., 2024) conditions on user-selected input spans. In such simple conditioning on the generation process, these methods depend on only a fixed window of past steps or inputs.

Constrained decoding is a steering method that has received recent attention from the natural language processing community (Scholak et al., 2021; Geng et al., 2023; Sakota and West, 2025; Nakshatri et al., 2025). It is unique among steering methods in that it explicitly models long-range and structurally complex patterns – such as programming language specifications, molecular structures, and privacy requirements (Ugare et al., 2025; Loula et al., 2025; Scholak et al., 2021; Poesia et al., 2022) – that cannot be captured by finite-window or input-fixed conditions. Formal grammars are central to this power.

Our contributions to this research direction are:

- We increase the expressive power of constrained decoders by designing a new formal grammar, Conditional Deontics over Terminals (CDoT), that is strictly more powerful than context-free grammars.
- We demonstrate that constrained decoding with LLMs under this new grammar formalism yields improvements on both a traditional text-to-SQL dataset and on a complex logic puzzle involving unit propagation.

2 Related Works and Design Goals

2.1 More Expressive Constrained Decoding

Constrained decoding with context-free grammars is sufficient for some use cases, such as enforcing JSON schemas on LLM outputs, where only the parent type and sibling types are needed to know what is allowed next in the tree-shaped output. However, context-free constraints fail to represent other use cases, such as ensuring that variables are declared before they are used, where the terminals of a variable-declaration subtree constrain what terminals are allowed in a separate variable-use subtree. Such cross-subtree terminal-to-terminal constraints are particularly challenging for constrained decoding, because the ancestor structures are unavailable when the LLM is still generating.

Consider figure 1, where the next terminals to be generated are intended to be constrained by a

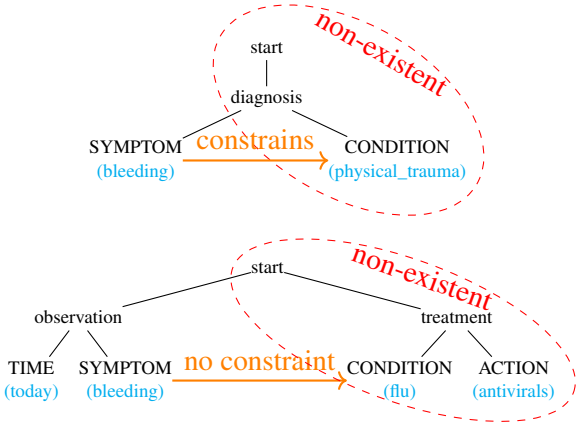

Figure 1: Example of enforcing a constraint where the terminals of a `SYMPTOM` constrain the terminals that may be generated for a `CONDITION` under a shared `diagnosis`. Because constrained decoding is incremental, the ancestors of a `CONDITION` are not known when it is being generated, making it difficult to decide whether the constraint should be applied (top) or not (bottom).

preceding `SYMPTOM` if they are eventually parsed into a `CONDITION` under the `SYMPTOM`'s *diagnosis* (top) but not not if they are eventually parsed into a `CONDITION` under another subtree (bottom). At the time that the terminals are generated, the ancestors have not been generated and thus it is non-trivial to ensure such constraints are satisfied. Prior work, such as in PiCARD (Scholak et al., 2021) and partially inherited by IterGen (Ugare et al., 2025), would ignore the ancestor part of the constraint, constraining the terminals even in the bottom case where the `CONDITION` is unrelated to the `SYMPTOM`. An alternative adopted by IterGen (Ugare et al., 2025) and GenLM Control (Loula et al., 2025) is to allow ancestor structures to be considered by invoking expensive backtracking and/or speculative look-ahead, where ancestor structures are allowed to emerge before constraints are applied. ChopChop (Nagy et al., 2026) employs a more powerful formalism to reason over partial syntax trees, but still relies on backtracking incorrect tokens. There have also been specialized solutions for particular context-sensitive constraints, such as indentation in Python syntax (Melcer et al., 2024).

2.2 Context-sensitive Grammar

While context-free grammars cannot represent cross-subtree references, some context-sensitive grammars support such references. Despite a diverse selection of mildly context-sensitive grammars (Joshi and Schabes, 1997; Vijay-Shanker and Weir, 1993; Okhotin, 2001; Mrykhin and Okhotin, 2023; Barash and Okhotin, 2014), such grammars are often not designed with the goal of efficient incremental prefix validation in mind. For example, Tree-Adjoining Grammars can be parsed non-incrementally with $O(n^6)$ step time-complexity, but incremental parsing requires the valid-prefix property, which increases complexity to $O(n^8)$ (Schabes, 1991). As another example, Barash and Okhotin (2014) propose a conjunctive grammar for matching variable uses to variable declarations, but it is restricted to variable names that all consist of the same letter and are only distinguished by the number of letter repetitions (see appendix C for further details).

2.3 Design Goals

The weaknesses of current constrained decoders and grammar formalisms motivates a new approach to constrained decoding with the design goals:

1. A grammar formalism with a cross-subtree reference mechanism that can be resolved automatically without requiring backtracking or speculative lookahead.
2. A parser for that grammar that can balance efficiency, expressive power, and practicality.

3 The CDoT Grammar Formalism

We introduce a new grammar formalism, Conditional Deontics over Terminals (CDoT), which we overview in the current section and formally define in appendix B.

CDoT extends context free grammars (CFGs; formally defined in appendix A) with *deontic rules*, which constrain not only the structure of derivations, but also the *obligations* and *permissions* of terminals. A *Conditional Deontics over Terminals (CDoT)* grammar is a pair $G^+ = (G, D)$ where

- $G = (V, \Sigma, R, S)$ is a CFG where:
  - V is a finite set of *nonterminal symbols*,
  - Σ is a finite set of *terminal symbols*, which may be regular expressions, with $V \cap \Sigma = \varnothing$,
  - $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *production rules* of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$,
  - $S \in V$ is the designated *start symbol*.
- D is a finite set of deontic rules, which constrain terminal instances generated by G.

Each deontic rule has an *anchor*, identifying the scope of the rule, and a *deontic function*, which

maps terminal instances to a *specification* pair $(\psi^{\Box}, \psi^{\Diamond})$, where $\psi^{\Box}$ is a set of regular expressions (necessary conditions) and $\psi^{\Diamond}$ is a set of regular expressions (permitted conditions).

Non-conditional deontic rules. The anchor of a non-conditional rule has the form

$$X^{con} \ll t^{con},$$

where X^{con} is a non-terminal ($X^{con} \in V$) and t^{con} is a terminal ($t^{con} \in \Sigma$). The semantics of a non-conditional deontic rule is that if its deontic function returns $(\psi^{\Box}, \psi^{\Diamond})$ as output, then all terminals descended from X must *collectively* satisfy every necessary condition ($r \in \psi^{\Box}$), and each individual terminal must match at least one permitted condition ($r \in \psi^{\Diamond}$).

Example (Balanced Meals). Given the grammar:

$$\begin{aligned} \textit{meal} &\rightarrow \textit{food}\ \ \textit{beverage} \\ \textit{food} &\rightarrow \texttt{DISH}\ \ \textit{food} \mid \texttt{DISH} \\ \textit{beverage} &\rightarrow \texttt{BEVERAGE}\ \ \textit{beverage} \mid \texttt{BEVERAGE}. \end{aligned}$$

The following deontic rule enforces that every meal includes grains, meat, and vegetables ($\Box$), while also allowing optional extras like `soup` or `steamed_egg` ($\Diamond$).

```
anchor: food ≪ DISH
def enforce_meal():
    return (□{rice|bread, chicken|fish,
              salad|roasted_vegetable},
            ◇{rice, bread, chicken, fish,
              salad, roasted_vegetable,
              steamed_egg, soup})
```

Conditional deontic rules. The anchor of a conditional rule has the form

$$t^{ant} \gg X^{ant} B \ldots D > X^{par} < X^{con} \ll t^{con}$$

where t^{ant} and t^{con} are terminals ($t^{ant}, t^{con} \in \Sigma$), and B, D, X^{ant}, and X^{con} are non-terminals ($B, D, X^{ant}, X^{con} \in V$) that appear in a parent production $X^{par} \rightarrow \ldots X^{ant} B \ldots D X^{con} \ldots$. The semantics of a conditional deontic rule is that if its deontic function returns $(\psi^{\Box}, \psi^{\Diamond})$ as output given the antecedent terminal instance t^{ant} as input, then all consequent terminals t^{con} must *collectively* satisfy every necessary condition ($r \in \psi^{\Box}$), and each individual terminal must match at least one permitted condition ($r \in \psi^{\Diamond}$).

Example (Food–Drink Pairings). Given the same grammar as in the previous example, the following deontic rule enforces that when a food and a beverage are paired within a meal, steaks require red wine, fish requires white wine, and spicy noodles require soda.

```
anchor: DISH ≫ food > meal < beverage ≪ BEVERAGE
def pair_food_drink(terminal_instance):
    match terminal_instance.text:
        case 'steak':
            return □{red_wine}, ◇{}
        case 'fish':
            return □{white_wine}, ◇{}
        case 'spicy_noodle':
            return □{soda}, ◇{}
```

Notice that, while a single antecedent variable *food* can cover multiple antecedent `DISH` variables, each invocation of the deontic function $match_drink_to_food$ takes a single instance of `DISH` as parameter. This prevents repetitive calling of the deontic function over different combinations. At the same time, the conjunctive nature of $\Box$ and disjunctive nature of $\Diamond$ will still naturally combine so that, if both fish and steak are ordered, both red and white wine are requested.

Each deontic rule can be further modified by attributes (appendix B.5).

3.1 Design Considerations

CDoT deviates from traditional formal grammars by allowing additional specifications to be attached to terminals. This is inspired in part by program variable declaration/use constraints, where we would like to constrain a code-generating LLM such that it can only generate the use of a variable `x` if has first generated the declaration of that variable, and thus the constraints must be expressed over real grammar terminals like the names of variables. See appendix C for a walkthrough of this problem, and section 5 for empirical demonstrations of the usefulness of CDoT for this and other problems. Attaching the specifications to terminals has the added advantage of aligning the cross-subtree reference mechanism with chart parsers, allowing for an efficient parser implementation (section 4.1).

Given $O(n^2)$ potential subtrees, an equal number of terminal specifications needs to be initialized and enforced in CDoT. Deontic logic provides (von Wright, 1951) an intuitive and efficient alternative. Instead of one specification per antecedent subtree, we initialize and evaluate one specification per antecedent terminal ($O(n)$ in total). The satisfaction of each antecedent subtree can be aggregated from terminals using simple on-GPU dynamic programming, where Necessary ($\Box$)" aggregates by conjunction, tightening like a lower bound, and Permitted ($\Diamond$)" aggregates by disjunction, loosening like

an upper bound (see appendix B and algorithm 1 for details). The specification itself is also designed for efficiency: a set of smaller regular expressions is faster to compile for the LLM tokenizer than a single large regular expression, and also yields a higher cache-hit rate.

4 Parsing CDoT Grammars

Our parser, like Jelinek and Lafferty (1991), is designed to support the valid-prefix property (i.e., finding errors incrementally and as soon as possible). It combines both top-down (Earley-like parsing) and bottom-up (CYK-like parsing) to maintain a parse chart that can incrementally add a single token in $O(n^2)$ time, resulting in $O(n^3)$ time for adding all n tokens. It enforces necessary conditions by denying the parent node from forming until necessary conditions are met, and enforces permitted conditions by allowing antecedent non-terminals to be used until a violating consequent terminal instance is committed. Full implementation details are in appendix E and code is available at `https://anonymous.4open.science/r/logit_parsers-034D`.

4.1 Cross-subtree References without Backtracking

A cross-subtree reference in CDoT takes the form of a conditional deontic rule anchor $t^{ant} \gg A > X < B \ll t^{con}$. Such a rule constrains an upcoming consequent terminal if:

- the antecedent non-terminal A has been entered into the parse chart
- parent X is reachable from top-down (Earley-like parsing)
- consequent B is reachable from bottom-up (CYK-like parsing)

Our parse chart allows direct access to all three properties, avoiding any need for backtracking or speculative lookahead to enforce deontic constraints when decoding with an LLM.

4.2 Safeness and Soundness

In contrast to PiCARD and IterGen (Scholak et al., 2021; Ugare et al., 2025), our implementation is safe in the sense that it would not accidentally prune away correct model output.

However, due to the decidability issue discussed in appendix B.4.2, we only achieve soundness for the strict-$\diamond$ semantics: such deontic rule instance is pruned only upon its first direct violation. At that point generation terminates if the grammar does not provide an alternative path. Such situations can be avoided by providing a fallback, such as in our Knight-and-Knave grammar (figure 2), where we separate derivations for good guesses and bad guesses (which trigger different deontic rules).

Furthermore, under union-$\diamond$ semantics, looser than specified constraints may result under the conditions that two deontic rules share at least one potential derivation tree, and each can also belong to derivation trees that exclude the other. In such cases, permission granted in one derivation tree may incorrectly propagate to another, as demonstrated in appendix D. Such situations can be avoided by only defining deontic rules that don't share derivation trees.

4.3 GPU Implementation

Our current implementation is CPU bound and the token-to-token time is similar to the HuggingFace implementation of Llama3 3B models. However, an efficient implementation of incremental CYK recognizer steps at around 100 µs with $n = 100$ and $|G^{CNF}| = 3000$ using a single stream multiprocessor (out of > 100) on a Nvidia H100 GPU.

A profiling of our program shows long GPU idle time because our implementation uses mostly short-lived GPU kernels. This can be addressed by techniques such as CUDA Graph or kernel fusion.

5 Experiments and Results

The goal of our experiments will be to demonstrate that the additional theoretical expressiveness of CDoT translates into practically useful expressiveness. (See appendix C for an example of how theoretically expressive conjunctive grammars fail to handle the practical problem of variable scoping.) We will thus validate the applicability and efficacy of our constrainer on different tasks.

We experiment on two datasets from recent natural language processing research: Knight and Knave (Xie et al., 2025) and Spider (Yu et al., 2018). These datasets represent points along a spectrum from synthetic data with complex constraints to non-synthetic data with simpler constraints. Knight and Knave is a synthetic dataset that requires complex reasoning with deep logical constraints, essentially asking LLMs to reason over logical forms in a similar manner to how LLMs are often asked to reason over programming questions. Spider is a non-synthetic dataset based on a translation task

with shallower constraints, asking LLMs to convert English questions to SQL, a task which has been widely used in constrained decoding research.

5.1 Knight and Knave

Knight and Knave (Xie et al., 2025) is a logic puzzle dataset where each character is either a knight (always tells the truth) or a knave (always lies). The task is to infer each character's type from their statements. For example, if Alice says "Alice and Bob have the same identity", and Bob says "Alice is lying", we must deduce that Alice is a knave and Bob is a knight. The dataset consists of textual and logical forms of puzzles with 2-8 characters as input, and character type assignments as output.

5.1.1 Constrainer Design

Figure 2 shows the context free grammar we employ for the output of the LLM on the Knight and Knave problem. It ensures that the model outputs a sequence of guesses. Each guess assigns roles to characters and boolean values to auxiliary variables. The grammar alone will not ensure the desired behavior, we must add deontic rules to achieve that.

To ensure that *sufficient_assignment_seq* is recognized iff an *assignment_seq* commits to at least one literal from each clause, we add the deontic rule:

```
anchor: sufficient_assignment_seq ≪ ASSIGNMENT
% attr: hidden_permitted, strict − ◇
def correct_guess():
    return (□{"x_i|...|x_j"
                for x_i ∨ ··· ∨ x_j
                in input_clauses},
            ◇{})
```

We add a similar rule for constraining *contradictory_assignment_seq*.

To ensure that a REPEATED_ASSIGNMENT is recognized iff it replicates a previous ASSIGNMENT within this guess, we add the deontic rule:

```
anchor: ASSIGNMENT >> assignment_seq_semi_1
        > repetition_assignment_seq
        < repeating_assignment
        ≪ REPEATED_ASSIGNMENT
% attr: strict − ◇
def repetition(terminal):
    return (□{}, ◇{terminal})
```

We then need deontic rules to perform *unit propagation*, an algorithm that deduces assignments based on committed assignments of other literals. To support this algorithm, we first convert the input logical statements into *conjunctive normal form*, where each logical statement is a conjunction of *clauses*, each clause being a disjunction of *literals* (a variable or its negation). For example, $(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_5)$. An arbitrary boolean expression can be converted into equalsatisfiable conjunctive normal form efficiently via the Tseitin transformation (Tseitin, 1983) by adding $O(n)$ auxiliary variables.

Unit propagation then exploits the fact that each clause in conjunctive normal form must have at least one literal to be true. Therefore, if all but one literal in a clause are false, the last remaining literal must be true. In the example above, if we assume x_1 is true, then x_3 must also be true, because it is the last remaining option to keep the second clause satisfied. Deontic rules for implementing the unit propogation algorithm are defined for each x_k in the input clauses. We add context-free rules of the form:

unit_prop → *x_k_condition semi_2*
x_k_assignment

We add deontic rules of the form:

```
anchor: x_k_assignment ≪ ASSIGNMENT
% attr: self_sufficient_necessary, strict − ◇
def x_k_condition():
    return (□{"x_k"}, ◇{})
```

For each input clause c that contains x_k, we add a deontic rule:

```
anchor: x_k_condition ≪ ASSIGNMENT
% attr: hidden_permitted, strict − ◇
def x_k_assignment():
    return (□{"NEGATE_LITERAL(x_m)"
                for x_m in c if m ≠ k},
            ◇{"x_m"
                if x_m exists in any clause
                and m ≠ k})
```

These deontic rules result in the real-time triggering of unit propagation and the real-time detection of good and bad guesses. As the LLM commits to literals, unit propagation will trigger immediately upon conditions being met, and will constrain generation to only entailed literals. Bad guesses will be detected immediately upon their first contradiction and generation will be forced to start a new guess. Good guesses will be detected immediately once the formula is solved and generation will be immediately terminated.

5.1.2 Experimental Setup and Baselines

We apply LLMs in a 0-shot setting, providing both the textual and logical forms of the puzzle in the input prompt and expecting assigments of characters to types as output.

We only compare against unconstrained models for the lack of alternatives. PiCARD is hard-coded

```
start                      →  bad_guess* good_guess
good_guess                 →  sufficient_assignment_seq semi_3 GOOD_GUESS_TAIL
sufficient_assignment_seq  →  assignment_seq
bad_guess                  →  contradictory_assignment_seq semi_3 CONTRADICTORY_GUESS_TAIL
                           |  repetition_assignment_seq semi_3 REPETITION_GUESS_TAIL
contradictory_assignment_seq →  assignment_seq
repetition_assignment_seq  →  assignment_seq_semi_1 repeating_assignment
assignment_seq_semi_1      →  assignment_seq semi_1
assignment_seq             →  assignment_seq semi_1 assignment
                           |  GUESS_HEAD assignment
                           |  unit_prop
assignment                 →  ASSIGNMENT
ASSIGNMENT                 →  (NAME "->" KK)
                           |  AUX_VAR
repeating_assignment       →  REPEATING_ASSIGNMENT
REPEATING_ASSIGNMENT²      →  ASSIGNMENT
semi_3                     →  SEMI_3 "\n"
semi_2                     →  SEMI_2 "\n"
semi_1                     →  SEMI_1 "\n"
SEMI_3³                    →  ";"
SEMI_2²                    →  ";"
SEMI_1¹                    →  ";"
GUESS_HEAD                 →  "Here is a different and improved guess: {{\n"
GOOD_GUESS_TAIL            →  "}} «DONE»"
CONTRADICTORY_GUESS_TAIL   →  "}} contradiction\n\n"
REPETITION_GUESS_TAIL      →  "}} Bad guess\n\n"
NAME                       →  /(Jacob|Noah|Michael|Liam|Ella|. . . |Ava)/
KK                         →  "knight" | "knave"
```

Figure 2: Context free grammar for LLM generations in the Knight and Knave problem.

Model	Constrained	Unconstrained		
	Thinking	*Non-thinking*	*Thinking*	
	512	*512*	*512*	*2048*
Llama-3.2-1B-Instruct	0.52	0.04	0.04	OOM
Llama-3.2-3B-Instruct	0.65	0.13	0.13	OOM
Qwen/Qwen3-0.6B	0.61	0.07	0.07	0.36
Qwen/Qwen3-1.7B	0.69	0.03	0.03	0.55

Table 1: Knight and Knave test: success rate of constrained vs. unconstrained models with 512 or 2048 token limits. Llama-3 models do not have a thinking switch, but does exhibit chain of thought-like behavior. Xie et al. (2025) reported 0.14 for Llama-3-8B-instruct in the same setting.

to SQL grammar. IterGen can only backtrack an incorrect generation, while our constraints proactively force an assignment once it is fixed, and it is unclear how many assignments should be backtracked once a conflict arises. ChopChop has similar backtracking issues and is not publicly released.

5.1.3 Results

Table 1 shows that our CDoT constrainer yields major improvements in LLM generation for the Knight and Knave problem. When LLM output is limited to 512 tokens[1], the constrained model success rate is ∼50 points higher than that of unconstrained models. When unconstrained models are allowed to generate 4 times more tokens (2048 tokens), constrained models with only 512 tokens still outperform them.

Figure 3 plots the number of output tokens for successful (dark) and failed (light) LLM responses across constrained (blue) and unconstrained (orange) Qwen3-1.7B. The constrained model solves a larger portion of puzzles with fewer tokens. The unconstrained model forms a flat distribution with many more output tokens required to solve puzzles.

Figure 4 shows that the constrained model can often solve the puzzle on the first attempt. This indicates that the constrainer and LLM are organically collaborating. If the constrainer alone was driving success, we would expect the distribution of guesses to be closer to random uniform, and it would be rare to solve the puzzle on the first guess.

Figure 8 in appendix F breaks down the success rate by the number of people in the puzzle, which is the measure of puzzle complexity built into the dataset by Xie et al. (2025). Constrained mod-

[1] In preliminary experiments, the constrained model began to loop after around 512 tokens, failing to solve additional problems as the token limit was raised.

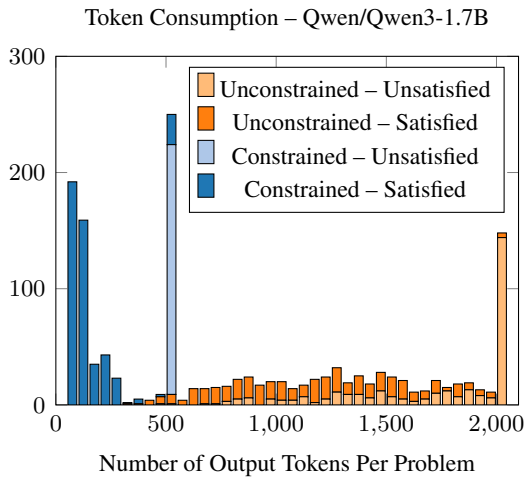

Figure 3: Knight and Knave test: token efficiency of successes (dark) vs failures (light) of constrained (blue) vs. unconstrained (orange) models. Constrained models were cut off at 512 tokens, unconstrained at 2048 tokens.

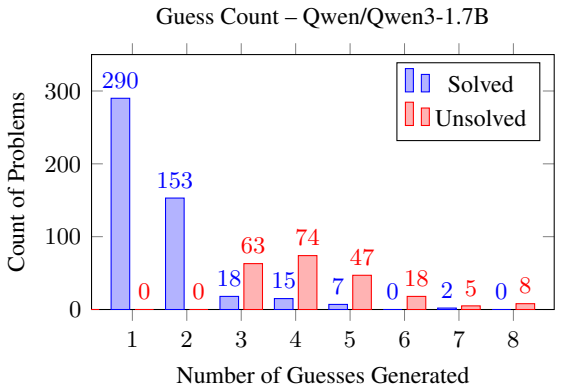

Figure 4: Knight and Knave test: constrained models solve many problems in the first guess. Notice that the success rate at first guess is significantly higher than random guess success rate, indicating organic LLM/Constrainer collaboration instead of constrainer dominance.

els from 0.6B to 3B parameters are still achieving success rates above 0.40 on the hardest 8-person puzzles, while the best of the unconstrained models does not even reach a success rate of 0.20 on these problems.

5.2 Text-to-SQL

Spider (Yu et al., 2018) is a text-to-SQL task, where the model is provided with a database schema and a natural text question as input, and it is expected to produce as output the SQL query to the given database that corresponds to the given question. Spider is commonly used as an evaluation dataset in constrained decoding research (Scholak et al., 2021; Jivani et al., 2025, etc.).

5.2.1 Constrainer Design

We use a grammar extracted from the SQLite source code as the base context-free grammar, then add constraints that allow only identifiers that come from databases containing the columns in the "SELECT" clause. We also run experiments with the additional constraint of limiting identifiers in the "SELECT" clause to the scope of all databases.

Formally, given an SQL grammar with rule:

$$select \rightarrow select_clause\ from_clause\ \ldots$$

we define the following deontic rules:

```
anchor: ID ≫ select_clause>select<from_clause ≪ ID
% attr: union − ◇
def id_in_database(terminal):
    return (□{},
            ◇{name if name is an
              identifier of a database
              containing terminal})
```

```
anchor: select_clause ≪ ID
% attr: union − ◇
def id_in_tables_or_columns():
    return (□{},
            ◇{name if name is a table or
              column of any database})
```

5.2.2 Experimental Setup and Baselines

We apply LLMs after fine tuning on the text-to-SQL training set following prior work (Scholak et al., 2021; Jivani et al., 2025). We provide both the database schema and the natural language question as input and expect the corresponding SQL code as output. We apply the constrainer to the LLM during both finetuning and inferencing. For both constrained and unconstrained models, we remove tokens from the LLM's tokenizer that do not align with SQL grammar terminals, such as ".$Name$" and "');". LLM performance is known to suffer when such tokens are not handled by the constrainer (Beurer-Kellner et al., 2024).

We again compare against unconstrained models due to lack of alternatives. PiCARD was implemented for T5 and is now defunct. IterGen and similar backtracking-based systems cannot be used to influence fine-tuning, as their constraints operate external to the LLM, rather than producing a logit mask as in our CDoT parser.

5.2.3 Results

Figure 5 shows that our CDoT constrainer yields improvements for 3 of the 4 models tested. Qwen 0.6B, Llama 1B, and Llama 3B all benefit from constrained decoding with 5 or more epochs. Qwen 1.7B does not benefit from the constrainer.

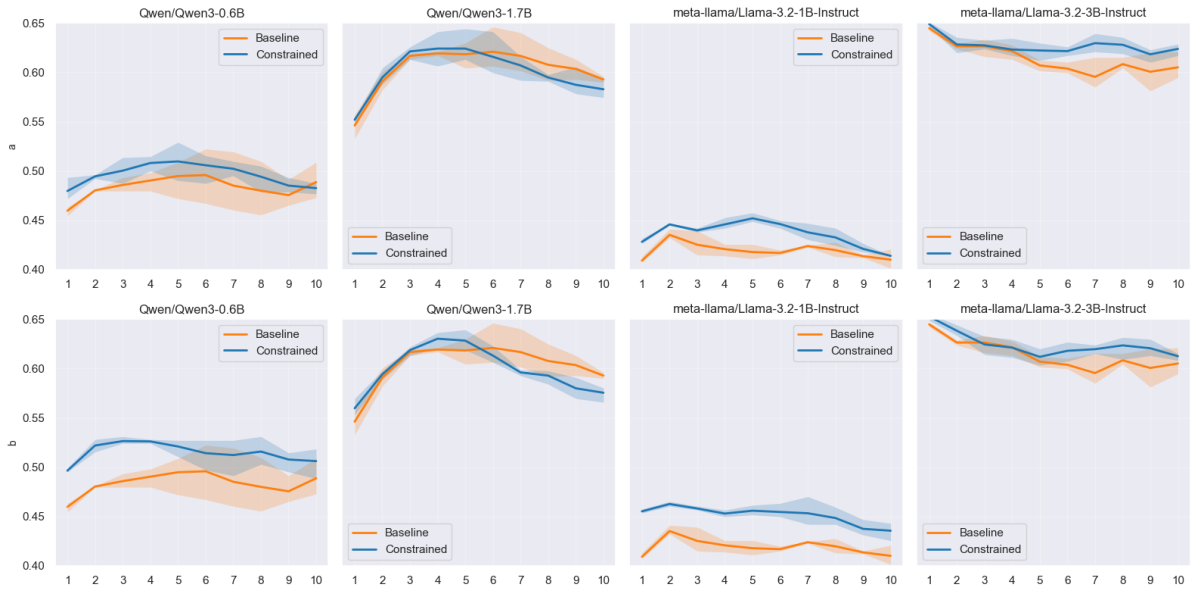

Figure 5: Text-to-SQL test: Execution accuracy of constrained vs unconstrained models over 3 runs and 10 epochs. Row (a) uses the `id_in_database` constraint defined in section 5.2.1. Row (b) additionally uses the `id_in_tables_or_columns` constraint. Constrained decoding helps three of the four models.

While we cannot directly compare to IterGen as noted above, we can observe that for Llama-3.2-1B, our fine-tuned approach yields a gain of ∼5 points in accuracy, from ∼40 to ∼45, similar to the ∼5 points gain in accuracy, from 25.5 to 30.9, reported by IterGen for a non-fine-tuned Llama-3.2-1B.

Error Analysis We conduct further analysis on the most and least benefited model (Llama 3.2 1B vs Qwen 3.1 1.7B) at 10 epochs of training with both with and without constrains enabled. We report the number of datum averaged over three runs.

Column name errors are divided into two types:

1. does not contain tuple alias: reduced in constrained case for both models ($135.0 \rightarrow 114.0$ for Llama, $60.3 \rightarrow 40.0$ for Qwen). This is expected as our constraints belong to this category. The remaining errors contains mostly "WHERE" clause using columns not in selected tables. When we try to model this relationship, the model performance is reduced.
2. contain tuple alias (e.g. generate "T1.NAME", but alias "T1" does not contain column "NAME"): worsens in constrained models ($150.3 \rightarrow 174.3$ for Llama, $102.0 \rightarrow 134.7$ for Qwen). CDoT cannot model this constrain on unmodified SQL grammars.

Constrained models also exhibit fewer syntax errors ($45.3 \rightarrow 0.75$ for Llama, and $5.3 \rightarrow 0.7$ for Qwen). Although deontic rules do not model syntax, the context-free skeleton alone would be insufficient: since SQL syntax permits identifiers in nearly every position, the constraint mask would be trivially all-true without the deontic rules.

We also attempted a "necessary" rule blocks the "FROM" clause from completion until sufficient tables are generated. We found this rule causes model to stuck in the loop during inference.

These observations indicate an complicated interaction between constrainer and model during finetuning. A more thorough analysis is out of scope of this work, and we did not conduct training hyperparameter search. However, both deontic rules we kept are active on more training tokens, thus we suspect training data size being a factor.

6 Conclusion

We exploit chart-based parsing to design a novel cross-subtree reference mechanism that achieves mild context-sensitivity. By allowing predicates to be attached to terminals, we expand the expressiveness. Attaching deontic functions to grammar rules improves both expressiveness and usability. Our parser's efficiency builds on prior work in GPU-accelerated parsing.

We use our formalism to implement the unit propagation algorithm, guiding reasoning traces on Knight-and-Knave puzzles and achieve significant performance gain at reduced token budget. Our constrainer also proves helpful in text-to-SQL.

Limitations

Formalism Design Our system makes certain assumptions on the underlying context-free skeleton grammar and deontic function. The salient aspects are discussed in section 4.2, appendix B.4.2, and appendix D. Additionally, our current implementation also fall backs to looser-than-specified approximation when there are sibling subtrees between the antecedent and consequent subtrees. Although the constrainer will not unexpectedly deny legal model outputs, it still creates challenge in practical engineering. A better understanding of these assumptions can be used to further refine to formalism to prevent or indicate potential violation. This can be achieved by either theoretical analysis or a wider study of potential use cases.

Formalism Implementation Our parser remains rudimentary and underperforming. The GPU idling issue discussed in section 4.3 requires immediate attention, and it is unclear whether SIMD-enabled CPUs is a better alternative solution to this problem. Additionally, our implementation does not yet exploit advanced hardware features such as shared memory. Finally, the $O(n^2)$ persistent parser state may become problematic for long generations.

Experiments Our experiments are far from a comprehensive coverage of constrained decoding applications. The interplay between constrainer and LLM also remains underexplored, with existing observations yielding conflicting outcomes: the constrainer and LLM cooperate organically out-of-the-box in the Knight-and-Knave experiment, yet performance degrades under certain constraints when fine-tuning on the Spider dataset.

References

Mikhail Barash and Alexander Okhotin. 2014. An extension of context-free grammars with one-sided context specifications. *Information and Computation*, 237:268–293.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding llms the right way: fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Zifeng Cheng, Jinwei Gan, Zhiwei Jiang, Cong Wang, Yafeng Yin, Xiang Luo, Yuchen Fu, and Qing Gu. 2025. Steering when necessary: Flexible steering large language models with backtracking. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Frederick Jelinek and John D. Lafferty. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17(3):315–353.

Smit Jivani, Saravam Maheshwari, and Sunita Sarawagi. 2025. Reliable answers for recurring questions: Boosting text-to-sql accuracy with template constrained decoding. *Proc. ACM Manag. Data*, 3(6).

Aravind K. Joshi and Yves Schabes. 1997. *Tree-Adjoining Grammars*, pages 69–123. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bruce W. Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. 2025. Programming refusal with conditional activation steering. *Preprint*, arXiv:2409.05907.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. 2025. Syntactic and semantic control of large language models via sequential monte carlo. In *The Thirteenth International Conference on Learning Representations*.

Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. 2024. Constrained decoding for fill-in-the-middle code language models via efficient left and right quotienting of context-sensitive grammars. *Preprint*, arXiv:2402.17988.

Mikhail Mrykhin and Alexander Okhotin. 2023. The hardest language for grammars with context operators. *Theoretical Computer Science*, 958:113829.

Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. 2026. Chopchop: A programmable framework for semantically constraining the output of language models. *Proc. ACM Program. Lang.*, 10(POPL).

Nishanth Sridhar Nakshatri, Shamik Roy, Rajarshi Das, Suthee Chaidaroon, Leonid Boytsov, and Rashmi Gangadharaiah. 2025. Constrained decoding with speculative lookaheads. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4681–4700, Albuquerque, New Mexico. Association for Computational Linguistics.

Alexander Okhotin. 2001. Conjunctive grammars. *J. Autom. Lang. Comb.*, 6(4):519–535.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *Preprint*, arXiv:2201.11227.

Marija Sakota and Robert West. 2025. Combining constrained and unconstrained decoding via boosting: BoostCD and its application to information extraction. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18356–18371, Suzhou, China. Association for Computational Linguistics.

Yves Schabes. 1991. The valid prefix property and left to right parsing of Tree-Adjoining Grammar. In *Proceedings of the Second International Workshop on Parsing Technologies*, pages 21–30, Cancun, Mexico. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg.

Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. 2025. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*.

K Vijay-Shanker and David J. Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636.

G. H. von Wright. 1951. Deontic logic. *Mind*, 60(237):1–15.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. 2025. On memorization of large language models in logical reasoning. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 2742–2785, Mumbai, India. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. 2024. Tell your model where to attend: Post-hoc attention steering for LLMs. In *The Twelfth International Conference on Learning Representations*.

A Context-Free Grammars and Languages

A *context-free grammar (CFG)*, which is a 4-tuple $G = (V, \Sigma, R, S)$ where:

- V is a finite set of *nonterminal symbols* (also called variables), which represent syntactic categories that can be expanded,
- Σ is a finite set of *terminal symbols*, which are the basic alphabet symbols that appear in the strings of the language, with $V \cap \Sigma = \varnothing$,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *production rules* of the form $A \to \alpha$, where $A \in V$ (a nonterminal) and $\alpha \in (V \cup \Sigma)^*$ (a string of terminals and/or nonterminals),
- $S \in V$ is the designated *start symbol*.

The language generated by G is defined as $L(G) = \{\, w \in \Sigma^* \mid S \Rightarrow^* w \,\}$, where $\Rightarrow^*$ denotes the reflexive, transitive closure of the single-step derivation relation $\Rightarrow$ induced by R. A language $L \subseteq \Sigma^*$ is called *context-free* if there exists a CFG G such that $L = L(G)$.

B Formal Semantics of CDoT

B.1 Basic Notations

Terminal Instance (t_j) is a subspan of the generation process. Each terminal instance is defined by a sequential index (j), a terminal type, and a user-defined payload. In the case of constrained decoding, a terminal instance might have type "VAR_NAME" and payload of token sequence ["start", "_", "time"].

Type-based Filtering ($[t_i \dots t_j]_b$) extracts the set of all terminal instances within the span that have terminal type b in sequence $t_i \dots t_j$. Formally,

$$[t_i \dots t_j]_b \triangleq \{t_k \mid i \leq k \leq j \wedge \text{type}(t_k) = b\}$$

Predicate (ρ) We write $\rho(t_i)$ to indicate that the single terminal instance t_i satisfies property ρ. For example, regular expressions are good predicate for constrained decoding (i.e. a variable name must start with a certain prefix). Alternatively, a neural probe or other heuristics might also be predicates.

Predicate Verifier A predicate verifier is needed to confirm a terminal instance satisfies the predi-

cate, for example, matching regular expressions or comparing to a threshold value.

Predicate Enforcer A predicate enforcer produces a new satisfying terminal instance, for example, running a regular expression constrainer or cutting off values. An all-permitting enforcer is acceptable and appropriate if permitted is set to permit-all (e.g., when using a neural probe predicate). Enforcers need to be mergable, with the merged enforcer producing only t_i s.t. $\rho_1(t_i) \vee \rho_2(t_i)$. Enforcers needs to be at least sound, if completeness is not achievable.

B.2 Terminal Specification

CDoT deviates from traditional formal grammars and allows grammar rules to specify properties to terminal instances under their scope.

Terminal Specification (ψ) is a set of predicates. Terminal $t_i \dots t_j$ satisfies a specification ψ is written as $t_i \dots t_j \in \psi$. Notice that the $\in$ operator is overloaded as specification is a set of predicates.

Necessary Specification ($\psi^\Box$) is the conceptual "lower bound". Each predicate in a necessary specification must be satisfied by at least one consequent terminal instance, formally

$$t_i \dots t_j \in \psi^\Box \iff \forall \rho \in \psi^\Box : \exists k \in [i, j] : \rho(t_k)$$

Permitted Specification ($\psi^\Diamond$) is the conceptual "upper bound". Each consequent terminal instance must satisfy at least one predicate in the permitted specification, formally

$$t_i \dots t_j \in \psi^\Diamond \iff \forall k \in [i, j] : \exists \rho \in \psi^\Diamond : \rho(t_k)$$

Obligation Implies Can we stipulate that every predicate in a necessary specification is automatically added to the matching permitted specification.

Aggregating Specifications is achieved by the union $\psi_A \cup \psi_B$. Intuitively, the necessary specification tightens (more predicates to be satisfied), and the permitted specification loosens (more predicates to choose from).

B.3 Deontic Function

A *deontic funtion* maps a single terminal instance to a token specification. This serves a similar role as functions in recursive descent parsers or IterGen. Invoking unrestricted functions during parsing increases both expressiveness and convenience, allowing, for example, a query to an external tool or a recursive LLM call during generation.

We define the function to take a single terminal instance as parameter, rather than all antecedent instances to reduce the number of deontic function calls from $O(n^2)$ to $O(n)$. We compute each terminal's deontic result once upon completion, then terminal instance can be discarded and save space if payload are large such as LLM hidden states. Due to the aggregation behavior of necessary and permitted semantics described in appendix B.2, taking a single terminal instance often gets the same result specification.

B.3.1 Notations of deontic function

Concretely, we write a deontic function as a python function that takes a terminal instance and returns a pair of necessary and permitted specifications as a tuple. Given a deontic function, we write $\Box t_i$ or $\Diamond t_i$ to indicate the necessary or permitted specification obtained by applying such deontic function to t_i. We introduce shorthands:

$$\Box(t_i \dots t_j) \triangleq \Box(t_i) \cup \Box(t_{i+1}) \cup \dots \cup \Box(t_j)$$
$$\Diamond(t_i \dots t_j) \triangleq \Diamond(t_i) \cup \Diamond(t_{i+1}) \cup \dots \cup \Diamond(t_j)$$

B.4 The Inference Rules

Terminal Instance Partitions (F, P, R) Under a CDoT subtree, each terminal instance belongs to one of three partitions. F are free terminal instances not under deontic constraints (e.g., keyword $SELECT$ in a SQL query is likely unconstrained). P are the permitted terminal instances under deontic constraints and receive permission ($\Diamond$) from the current subtree. R are the remaining terminal instances under deontic constraints, but which have no permission received from the current subtree (e.g., an invocation of variable from outer scope).

Assertion ($A(t_i \dots t_j \mid \langle R, F \rangle)$) is read as $t_i \dots t_j$ has property A, conditioned on the terminal instances in $R \subseteq t_i \dots t_j$ receiving permission from ancestors. We say $t_0 \dots t_n$ is in the language of grammar if and only if $START(t_0 \dots t_n \mid \varnothing)$.

B.4.1 Context-Free Rules

We rewrite context-free rules into Chomsky normal form to include the deontic condition.

Terminal Rule $X \to a$ generates a single terminal from a non-terminal. Because it is not under deontic constraint yet, t_i belongs to partition F.

$$\frac{\text{type}(t_i) = a}{X(t_i \mid \langle \emptyset, \{t_i\} \rangle)} \qquad \text{(Terminal)}$$

Context-free Rule $X \to A\ B$ simply aggregates and passes on the deontic condition.

$$\frac{A(t_i \dots t_j \mid \langle R_A, F_A \rangle) \quad B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle)}{X(t_i \dots t_k \mid \langle R_A \cup R_B, F_A \cup F_B \rangle)} \text{(Context-free)}$$

Since neither rule is conditioned on deontic premises nor adds terminals to the R partition, CDoT behaves identically to a context-free grammar in the absence of deontic rules.

B.4.2 Strict Permitted Rule

The crux of the problem is defining behavior when deontic rules are nested. We begin with a naive but useful semantics termed "strict permitted."

Strict-$\Diamond$ rule requires the consequent to strictly satisfy the terminal specification returned by the antecedent's deontic function, disregarding any constraints from ancestor rules or rules nested within the consequent. As a result, all terminals of consequent type (b) under the consequent non-terminal (B) belong to partition P for parent X, and therefore do not appear in F_x or R_x, as shown in the inference rule.

$$\frac{\begin{array}{c} A(t_i \dots t_j \mid \langle R_A, F_A \rangle)\ B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle) \\ [t_{j+1} \dots t_k]_b \in \Box([t_i \dots t_j]_a) \\ [t_{j+1} \dots t_k]_b \in \Diamond([t_i \dots t_j]_a) \end{array}}{\begin{array}{c} X(t_i \dots t_k \mid \langle R_X, F_X \rangle) \\ R_X = (R_A \cup R_B) \setminus [t_{j+1} \dots t_k]_b \\ F_X = (F_A \cup F_B) \setminus [t_{j+1} \dots t_k]_b \end{array}} \text{(Deontic - Strict } \Diamond)$$

Upon closer examination, one may realize that in a system with only context-free and strict-$\Diamond$, such system again keeps $R = \varnothing$ and never conditions on $\langle R, F \rangle$. Thus, a CYK-like parser can be easily implemented if the entire terminal instance sequence is available.

However, decidability issues arise when we seek to parse incrementally while maintaining the valid-prefix property. Because each consequent terminal instance can further serve as an antecedent in descendant subtrees—used as inputs to another deontic function—the parser must determine whether future terminals could yield a terminal specification incompatible with the current one.

B.4.3 Union Permitted Rule

Beside tractability, the strict-$\Diamond$ semantic sometimes also does not capture desired expressions when we need to aggregate specifications from multiple antecedent subtrees, for example, variables from different scopes.

Union-$\Diamond$ rule While keeping the necessary($\Box$) behavior unchanged, we allow subtrees to constrain terminal instances but receive permission from ancestors. This is reflected in removal of $\Diamond$ condition in the premise and updated R_X definition. Notice that we are overloading the $\setminus$ operator between terminal instance collection and specification, $(R_B \cup F_B) \setminus \Diamond[t_i \dots t_j]_a \triangleq \{t \mid t \in [R_B \cup F_B]_b \wedge t \notin \Diamond[t_i \dots t_j]_a\}$.

$$\frac{\begin{array}{c} A(t_i \dots t_j \mid \langle R_A, F_A \rangle)\ B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle) \\ [t_{j+1} \dots t_k]_b \in \Box([t_i \dots t_j]_a) \end{array}}{\begin{array}{c} X(t_i \dots t_k \mid \langle R_X, F_X \rangle) \\ R_X = R_A \cup (R_B \cup F_B \setminus \Diamond[t_i \dots t_j]_a) \\ F_X = (F_A \cup F_B) \setminus [t_{j+1} \dots t_k]_b \end{array}} \text{(Deontic - Union } \Diamond)$$

B.5 Deontic Rule Attributes

Additional attributes can be attached to each deontic rule to customize beahvior.

Strict and Union $\Diamond$: described above

self_sufficient_necessary: When multiple deontic rules are share the same anchor, the default behavior is require all necessary rules to be satisfied. However, a self-sufficient necessary rule overrides other necessary conditions if it is satisfied.

hidden_permitted: The permitted condition is only used to disqualify an anchor during generation, it does not impact LLM token mask next step.

C Theoretical Expressiveness Vs Practicalness

Conjunctive grammars achieve beyond-context-free expressiveness while keeping a tractable computation complexity of $O(n)$ to $O(n^3)$. It is also easy to envision a prefix validator with similar complexity (comparing to traditional mildly-context-sensitive such as tree-adjoint grammar at $O(n^9)$ (Schabes, 1991)). However, theoretical expressiveness does not necessarily translate into practical expressive power.

We demonstrate this aspect using example 2 from Barash and Okhotin (2014), which is a simplified task of verifying a variable name is declared before used, formally,

Conjunctive Grammar

var_list	$\rightarrow$ decl var_list $\mid \varepsilon$
	$\rightarrow$ matched_use var_list
decl	$\rightarrow a$ decl $\mid c$
use	$\rightarrow b$ use $\mid c$
matched_use	$\rightarrow$ use $\& \trianglelefteq$ any_var_list pair c
any_var_list	$\rightarrow$ decl var_list
	$\rightarrow$ use var_list $\mid \varepsilon$
pair	$\rightarrow a$ pair $b \mid c$ any_var_list

CDoT

var_list $\rightarrow$ var_list var | DECL

var $\rightarrow$ DECL | USE

DECL $\rightarrow a^*c$

USE $\rightarrow b^*c$

```
anchor:
  DECL ≫ var_list > var_list < var ≪ USE
def permit_passthrough(decl):
    return □{}, ◇{decl}
```

Figure 6: Side-by-side comparison of a conjunctive grammar from example 2 of Barash and Okhotin (2014) and an equivalent CDoT grammar. The conjunctive grammar implementation is not only more verbose, it also cannot be extended beyond unary encoding of variable name, make it unable to be used in practical case. CDoT on the other hand, uses a simple skeleton grammar, and the deontic function can leverage Python semantics to support more complex interactions between the antecedent and consequent.

$$L^{var} = \left\{ u_1 \dots u_n \;\middle|\; \begin{array}{l} \text{for every } u_i, \\ \textbf{either } u_i \in a^*c, \\ \textbf{or } u_i = b^k c \text{ and} \\ \text{there exists } j < i, \\ \text{for which } u_j = a^k c \end{array} \right\}$$

Each in-language string is a sequence of variable declaration (of form a^*c) and variable use (of form b^*c). A variable declaration and a variable use matches if and only if the number of a is equal to the number of b. A variable must be declared before first use. For example, $acbcaacbbc$ is a valid sequence that declares variable ac and aac before use, and $aacbc$ is an invalid sequence that uses bc before declaring ac. One can verify this language is beyond context-free by using $\{a^n b^m c^n d^m | m, n \geq 0\}$ for reduction.

Barash and Okhotin (2014) implements a conjunctive grammar for this language, shown on the left of figure 6. The centerpiece of this grammar is rule "matched_use $\rightarrow$ use $\& \trianglelefteq$ any_var_list pair c", which allows derivation "α matched_use $\Rightarrow \alpha$ use" only if "any_var_list pair $c \Rightarrow^* \alpha$ use". The pair non-terminal generates a pair of matching variable declaration and use.

We implement an equivalent grammar using CDoT on the right of figure 6. The skeleton grammar is a left-recursive list of variables. And the deontic function simply adds every variable declaration to the permitted set of variable uses.

Besides the obvious difference in verbosity between two grammars, there is a deeper problem in how variable names are encoded. The conjunctive grammar implementation does not offer a clear way to move beyond unary encoded names. CDoT on the other hand, can be switched to encode variable names in full natural language alphabet, we only need to define DECL and USE with appropriate regular expressions. Furthermore, full Python semantics is available in deontic function to allow more complex relationships to be implemented.

D Union-$\Diamond$ Counter Example

Under union-$\Diamond$ semantics, consider a prefix where two deontic rule instances satisfy the following conditions: (1) they share at least one potential derivation tree, and (2) each can also belong to derivation trees that exclude the other. In such cases, permission granted in one derivation tree may incorrectly propagate to another, causing cross contamination of permissions.

We illustrate this with two examples: one demonstrating correct pruning behavior, and one demonstrating a failure case.

D.1 Case 1: Sound and complete (as strict as specified)

Consider trees in figure 7. Assume a grammar allows tree 1 and tree 2, but does not have tree 3. Although node A is used in two separate trees, it does not cause problem. Tree 2 is pruned after committing t_2 for lack of permission. Tree 1 is kept alive for having A_2, but pruned after committing t_3

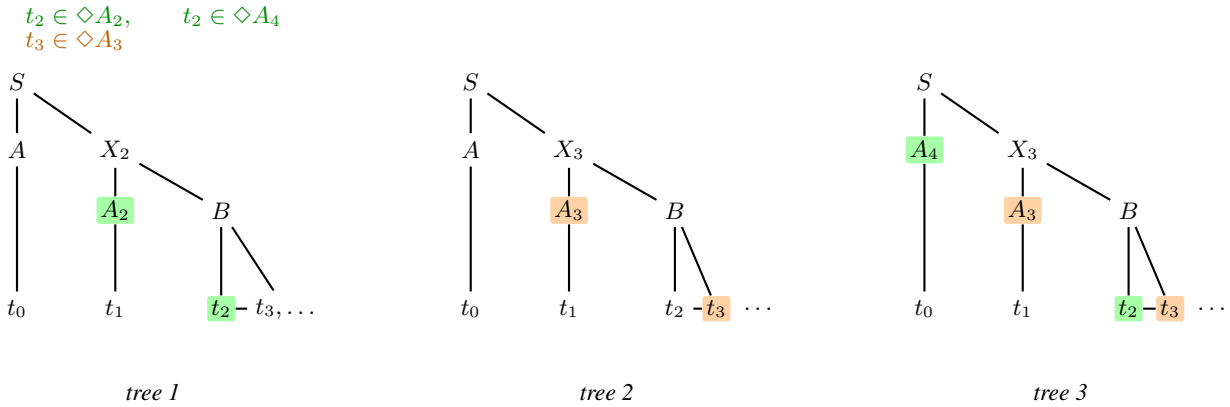

Figure 7: Three trees used in the counter example (highlighted nodes correspond to the committed constraints).

for lack of permission. This is illustrated in table 2.

	t_2	t_3
A	alive (prop. from A_2)	pruned
A_2	alive ($t_2 \in \Diamond A_2$)	pruned
A_3	pruned	—
tree 1	alive ($t_2 \in \Diamond A_2$)	pruned
tree 2	pruned	—

Table 2: Case 1 (sound and complete): after committing, both trees are pruned.

D.2 Case 2: Complete but not sound (looser than specified)

Now assume a grammar allows tree 1, tree 2, and tree 3. Tree 1 and tree 2 as we know should have been pruned after committing t_2 and t_3, but instead tree 2 is incorrectly kept alive. This is because node A is kept alive by being tree 1 and 2, node A_3 is kept alive by being in tree 3, collectively all components of tree 2 survives, as shown in table 3

	t_2	t_3
A	alive (prop. from A_2)	alive (prop. from A_3)
A_2	alive ($t_2 \in \Diamond A_2$)	pruned
A_3	alive (prop. from A_4)	alive ($t_3 \in \Diamond A_3$)
A_4	alive (prop. from A_3)	alive ($t_3 \in \Diamond A_4$)
tree 1	alive ($t_2 \in \Diamond A_2$)	pruned
tree 2	alive	alive
tree 3	alive ($t_2 \in \Diamond A_4$)	alive ($t_3 \in \Diamond A_3$)

Table 3: Case 2 (complete but not sound): propagation keeps some combinations alive that cannot co-occur in any single grammar-valid tree. Red boxes mark components falsely kept alive.

ERROR: no tree contains A, A_2, and A_3.

E Parser Details

Data Structure (LF$[tier, row, X \rightarrow A\ B]$**,** $\boldsymbol{TIER}[row, X]$) Chart-like parsers such as CYK use a table indexed by row, $column$ and non-terminal X. The incremental CYK recognizer uses the same table with the alternative index $tier$, row, X, where $tier = col + row$. The CDoT parser inherits the boolean matrix $\boldsymbol{TIER}[row, X]$, and the left-foot boolean tensor **LF**$[tier, row, X \rightarrow A\ B]$, with the last dimension indexed by binarized grammar rules.

Necessary (□) is enforced by denying the parent node from forming until the necessary condition is met. In the case that the underlying context-free grammar allows a finite number of consequent terminals, we assume the prescribed necessary requirement does not exceed this limit and the LLM completes the necessary requirement before the limit is exhausted. If this assumption is violated and the grammar does not provide an alternative path, generation will terminate.

Permitted (◇) enforcement allows an antecedent non-terminal A instance to be used in anchoring grammar rule $X \rightarrow A\ B$ until a violating consequent terminal instance is committed.

E.1 Preprocessing

After a new terminal instance t_n is completed, we need to find left feet that 1) have all necessary conditions **fulfilled** after accepting t_n, 2) can give permission to **enable** t_n. This can be achieved by simple dynamic programming shown as algorithm 1. We omit proof for simplicity.

$$
\begin{aligned}
&\forall j \leq \textsc{Fulfiller}[row, col] \Leftrightarrow \\
&\quad [t_j \ldots t_n]_b \in \Box\big([t_{col} \ldots t_{col+row}]_a\big) \\
&\textsc{Enabler}[row, col] \Leftrightarrow \\
&\quad t_n \in \Diamond([t_{col} \ldots t_{col+row}]_a)
\end{aligned}
\tag{1}
$$

E.2 Commit Batch

We update the parser for newly formed terminal instance t_n. We call the persistent state alive left foot, $\mathbf{LF_n^{ALIVE}}$. This requires a bottom-up and a top-down pass that can be run in parallel. The pseudo code can be found as algorithm 2 and algorithm 6.

E.3 Probe Batch

Probe wave finds all valid next terminal types and confirms each doentic left foot belongs to a derivation that supports current prefix $t_0 \ldots t_n$.

Additionally, to confirm each consequent terminal type have a unconstrained path, we run an additional bottom-up incremental CYK step with set of grammar rules excluding corresponding deontic anchors. If a terminal type has a unconstrained path, the constrainer receive its context-free definition as specification.

Shown as algorithm 10 and algorithm 11

E.4 Post Processing

If a antecedent terminal is under a valid deontic left feet, then $\psi^{\Diamond}$ resulting from the corresponding deontic function should be added to enforcer during generation of the consequent terminal (algorithm 13). This takes a simple top-down dynamic programming with $O(|D| \cdot n^2)$ complexity.

Algorithm 1: Deontic Preprocessing

Input: Terminal sequence $t_0 \dots t_n$, deontic rule with antecedent type a and consequent type b, necessary function $\Box$, permitted function $\Diamond$

Output: FULFILLED$[row, col]$, ENABLER$[row, col]$

```
// Base case: row 0
```

for $col \leftarrow 0$ **to** $n-1$ **do**

  if $\text{type}(t_{col}) \neq a$ **then**

    FULFILLED$[0, col] \leftarrow +\infty$;

    ENABLER$[0, col] \leftarrow \mathit{false}$;

  else

    $J \leftarrow \{\, j \mid [t_j \dots t_n]_b \in \Box(t_{col}) \,\}$;

    $J \leftarrow J \cup \{-\infty\}$;

    FULFILLED$[0, col] \leftarrow max(J)$;

    ENABLER$[0, col] \leftarrow t_n \in \Diamond t_{col}$;

```
// Inductive step:
```

for $row \leftarrow 1$ **to** $n-1$ **do**

  $L^{FF} \leftarrow$ FULFILLED$[row{-}1,\ :{-}1]$;

  $R^{FF} \leftarrow$ FULFILLED$[row{-}1,\ 1{:}]$;

  FULFILLED$[row] \leftarrow \min(L^{FF}, R^{FF})$;

  $L^{EB} \leftarrow$ ENABLER$[row{-}1,\ :{-}1]$;

  $R^{EB} \leftarrow$ ENABLER$[row{-}1,\ 1{:}]$;

  ENABLER$[row] \leftarrow L^{EB} \vee R^{EB}$;

Algorithm 2: Commit↑: Computing $\mathsf{LF}_n^{\mathsf{ALIVE}}$ (Bottom-Up)

Input : ENABLER, FULFILLER, $\mathsf{LF}_{n-1}^{\mathsf{ALIVE}}$, FIRST, t_n
Output : $\mathsf{LF}_n^{\mathsf{ALIVE}}$

```
// Initialization
```
$\mathsf{LF}_n^{\mathsf{ALIVE}} \leftarrow$ **false** everywhere
$\mathsf{PARENT}^{\mathsf{FREE}}, \mathsf{PARENT}^{\mathsf{CONSTR}} \leftarrow$ **false** everywhere
$\mathsf{ANCESTOR}^{\mathsf{FREE}}, \mathsf{ANCESTOR}^{\mathsf{CONSTR}} \leftarrow$ **false** everywhere

```
// Base Case: row = 0
```
foreach *terminal rule* $(X \rightarrow t_n)$ **do**
  $\mathsf{PARENT}^{\mathsf{FREE}}[0, X] \leftarrow \mathbf{true}$

foreach $V \in \mathcal{N}$ **do**
  $\mathsf{ANCESTOR}^{\mathsf{FREE}}[0, V] \leftarrow \mathsf{PARENT}^{\mathsf{FREE}}[0, V]$

AddAncestor(*ANCESTOR*$^{FREE}[0]$, *FIRST*)

```
// Inductive Step
```
for $row_X \leftarrow 1$ **to** n **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$
    $row_A \leftarrow row_X - row_B - 1$
    foreach *rule* $(X \rightarrow A\ B)$ **do**
      if $\neg$ $LF_{n-1}^{ALIVE}[tier_A, row_A, X \rightarrow A\ B]$ **then**
        continue
      $\mathtt{Merge}^{\uparrow}_{\mathtt{commit}}(row_X, row_B, tier_A, row_A, X \rightarrow A\ B)$
  AddAncestor(*ANCESTOR*$^{FREE}[row_X]$, *FIRST*)
  AddAncestor(*ANCESTOR*$^{CONSTR}[row_X]$, *FIRST*)

```
// Finalization: Create new left feet at tier n
```
for $row \leftarrow 0$ **to** n **do**
  foreach *rule* $(X \rightarrow A\ B)$ **do**
    $\mathsf{LF}_n^{\mathsf{ALIVE}}[n, row, X \rightarrow A\ B] \leftarrow \mathsf{PARENT}^{\mathsf{FREE}}[row, A] \vee \mathsf{PARENT}^{\mathsf{CONSTR}}[row, A]$

return $\mathsf{LF}_n^{\mathsf{ALIVE}}$

Algorithm 3: Merge$^{\uparrow}_{commit}$ (Context-Free)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \text{CF}$

if $\mathit{PARENT}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{FREE}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\mathit{PARENT}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\mathit{ANCESTOR}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{FREE}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\mathit{ANCESTOR}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 4: Merge$^{\uparrow}_{commit}$ (Strict Permitted)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \textsc{Strict}$

$\mathit{enabled} \leftarrow \mathsf{ENABLER}[tier_A, row_A, X \to A\ B]$
$\mathit{fulfilled} \leftarrow \mathsf{FULFILLER}[tier_A, row_A, X \to A\ B] \geq (n - row_B)$

`// All paths must satisfy ◇ at this scope`

if $\mathit{enabled} \wedge \big(\mathit{ANCESTOR}^{\mathit{FREE}}[row_B, B] \vee \mathit{ANCESTOR}^{\mathit{CONSTR}}[row_B, B]\big)$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\mathit{enabled} \wedge \mathit{fulfilled} \wedge \big(\mathit{PARENT}^{\mathit{FREE}}[row_B, B] \vee \mathit{PARENT}^{\mathit{CONSTR}}[row_B, B]\big)$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 5: Merge$^{\uparrow}_{commit}$ (Union Permitted)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \textsc{Union}$

$enabled \leftarrow \mathsf{ENABLER}[tier_A, row_A, X \to A\ B]$
$fulfilled \leftarrow \mathsf{FULFILLER}[tier_A, row_A, X \to A\ B] \geq (n - row_B)$

`// Already permitted: propagate without re-checking`
if $\mathit{ANCESTOR}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}^{\mathsf{ALIVE}}_n[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$
if $\mathit{PARENT}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}^{\mathsf{ALIVE}}_n[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

`// Free` t_n `entering deontic scope: check` $\diamond$
if $enabled \wedge \mathit{ANCESTOR}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}^{\mathsf{ALIVE}}_n[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$
if $enabled \wedge fulfilled \wedge \mathit{PARENT}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}^{\mathsf{ALIVE}}_n[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 6: Commit$\downarrow$: Computing $\mathbf{LF}^{\textsc{Alive}}$ (Top-Down)

Input: $\textsc{Enabler}, \textsc{Fulfiller}, \mathbf{LF}^{\textsc{Alive}}_{n-1}, \textsc{First}, t_n$
Output: $\mathbf{LF}^{\textsc{Alive}}_n$
`// Initialization`
$\mathbf{LF}^{\textsc{Alive}}_n \leftarrow$ **false** everywhere;
$\textsc{Child}^{\textsc{Free}}, \textsc{Child}^{\textsc{Constr}} \leftarrow$ **false** everywhere;
$\textsc{Ancestor}^{\textsc{Free}}, \textsc{Ancestor}^{\textsc{Constr}} \leftarrow$ **false** everywhere;

`// Base case: Start symbol at row` n
foreach $V \in \mathit{Nonterminals}$ **do**
  $\textsc{Ancestor}^{\textsc{Free}}[n, V] \leftarrow (V = S)$;
$\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Free}}[n], \textsc{First})$;

`// Inductive step: top-down`
for $row_X \leftarrow n$ **to** 1 **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \to A\,B)$ **do**
      if $\neg\,\mathbf{LF}^{\textsc{Alive}}_{n-1}[tier_A, row_A, X \to A\,B]$ **then continue**;
      $\textsc{Merge}\downarrow(row_X, row_B, tier_A, row_A, X \to A\,B)$;
  $\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Free}}[row_X - 1], \textsc{First})$;
  $\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Constr}}[row_X - 1], \textsc{First})$;
return $\mathbf{LF}^{\textsc{Alive}}_n$;

Algorithm 7: Merge↓ (Context-Free)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) = \text{CF}$
if $\textsc{Child}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Free}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $\textsc{Child}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $\textsc{Ancestor}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Free}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $\textsc{Ancestor}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 8: Merge↓ (Strict Permitted)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) = \textsc{Strict}$
$enabled \leftarrow \textsc{Enabler}[tier_A, row_A, X \to A\,B]$;
$fulfilled \leftarrow \textsc{Fulfiller}[tier_A, row_A, X \to A\,B] \geq (n - row_B)$;

`// All paths must satisfy ◇ at this scope`
if $enabled \wedge \big(\textsc{Ancestor}^{\textsc{Free}}[row_X, X] \vee \textsc{Ancestor}^{\textsc{Constr}}[row_X, X]\big)$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $enabled \wedge fulfilled \wedge \big(\textsc{Child}^{\textsc{Free}}[row_X, X] \vee \textsc{Child}^{\textsc{Constr}}[row_X, X]\big)$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 9: Merge↓ (Union Permitted)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) = \textsc{Union}$
$enabled \leftarrow \textsc{Enabler}[tier_A, row_A, X \to A\,B]$;
$fulfilled \leftarrow \textsc{Fulfiller}[tier_A, row_A, X \to A\,B] \geq (n - row_B)$;

`// Already constrained: propagate (permission granted below)`
if $\textsc{Ancestor}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $\textsc{Child}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

`// Free` t_n `entering deontic scope: check ◇, if satisfied → constrained`
if $enabled \wedge \textsc{Ancestor}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $enabled \wedge fulfilled \wedge \textsc{Child}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 10: Probe↑

Input: b – hypothetical terminal type at position n; $\mathbf{LF}^{\text{ALIVE}}$; FIRST
Output: $\mathbf{LF}^{\text{REACHABLE}\uparrow}$
$\text{ANCESTOR}^{\uparrow}[\cdot,\cdot] \leftarrow \textit{false}$;
$\mathbf{LF}^{\text{REACHABLE}\uparrow}[\cdot,\cdot,\cdot] \leftarrow \textit{false}$;
foreach *terminal rule* $(V \rightarrow b)$ **do**
  $\text{ANCESTOR}^{\uparrow}[0, V] \leftarrow \textit{true}$;
$\textsc{AddAncestor}(\text{ANCESTOR}^{\uparrow}[0], \text{FIRST})$;
for $row_X \leftarrow 1$ **to** n **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \rightarrow A\ B)$ **do**
      if $\mathbf{LF}^{\text{ALIVE}}[tier_A, row_A, X \rightarrow A\ B] \wedge \text{ANCESTOR}^{\uparrow}[row_B, B]$ **then**
        $\text{ANCESTOR}^{\uparrow}[row_X, X] \leftarrow \textit{true}$;
        $\mathbf{LF}^{\text{REACHABLE}\uparrow}[tier_A, row_A, X \rightarrow A\ B] \leftarrow \textit{true}$;
  $\textsc{AddAncestor}(\text{ANCESTOR}^{\uparrow}[row_X], \text{FIRST})$;
return $\mathbf{LF}^{\text{REACHABLE}\uparrow}$

Algorithm 11: Probe↓

Input: $\mathbf{LF}^{\text{ALIVE}}$; FIRST
Output: $\mathbf{LF}^{\text{REACHABLE}\downarrow}$
$\text{ANCESTOR}^{\downarrow}[\cdot,\cdot] \leftarrow \textit{false}$;
$\mathbf{LF}^{\text{REACHABLE}\downarrow}[\cdot,\cdot,\cdot] \leftarrow \textit{false}$;
$\text{ANCESTOR}_{\text{uncon}}[\cdot,\cdot,\cdot] \leftarrow \textit{false}$; `// indexed by` $[b, row, V]$
$\text{ANCESTOR}^{\downarrow}[n, \textit{START}] \leftarrow \textit{true}$;
$\textsc{AddDescendant}(\text{ANCESTOR}^{\downarrow}[n], \text{FIRST})$;
foreach *deontic consequent terminal* b **do**
  $\text{ANCESTOR}_{\text{uncon}}[b, n, \textit{START}] \leftarrow \textit{true}$;
  $\textsc{AddDescendant}(\text{ANCESTOR}_{\text{uncon}}[b, n], \text{FIRST})$;
for $row_B \leftarrow n - 1$ **to** 0 **do**
  for $row_X \leftarrow row_B + 1$ **to** n **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \rightarrow A\ B)$ **do**
      if $\mathbf{LF}^{\text{ALIVE}}[tier_A, row_A, X \rightarrow A\ B] \wedge \text{ANCESTOR}^{\downarrow}[row_X, X]$ **then**
        $\text{ANCESTOR}^{\downarrow}[row_B, B] \leftarrow \textit{true}$;
        $\mathbf{LF}^{\text{REACHABLE}\downarrow}[tier_A, row_A, X \rightarrow A\ B] \leftarrow \textit{true}$;
        foreach *deontic consequent terminal* b **do**
          if $(X \rightarrow A\ B)$ *is not anchor of any deontic rule with consequent* b **then**
            if $\text{ANCESTOR}_{\text{uncon}}[b, row_X, X]$ **then**
              $\text{ANCESTOR}_{\text{uncon}}[b, row_B, B] \leftarrow \textit{true}$;
  $\textsc{AddDescendant}(\text{ANCESTOR}^{\downarrow}[row_B], \text{FIRST})$;
  foreach *deontic consequent terminal* b **do**
    $\textsc{AddDescendant}(\text{ANCESTOR}_{\text{uncon}}[b, row_B], \text{FIRST})$;
return $\mathbf{LF}^{\text{REACHABLE}\downarrow}$

Algorithm 12: AddAncestor / AddDescendant

Procedure ADDANCESTOR($cell$, FIRST)
  foreach $V \in Nonterminals$ **do**
    if $\exists W : cell[W] \wedge \text{FIRST}[V, W]$ **then**
      $cell[V] \leftarrow true$;

Procedure ADDDESCENDANT($cell$, FIRST)
  foreach $W \in Nonterminals$ **do**
    if $\exists V : cell[V] \wedge \text{FIRST}[V, W]$ **then**
      $cell[W] \leftarrow true$;

Algorithm 13: Collect Permitted Constraints

Input : $\text{LF}^{\text{RRA}}[\text{tier}, \text{row}, \text{rule}]$: `bool`
Input : Terminal sequence $t_0 \ldots t_{n-1}$
Input : Deontic rules D
Output : $\psi_b^{\Diamond}$: permitted specification sent to enforcer

$\text{LF}^{\text{RRA}}[\text{tier}, \text{row}]$	$\text{Reachable}^{\uparrow} \wedge \text{Reachable}^{\downarrow} \wedge$ Alive for each left foot
$\text{memo}[\text{row}, \text{col}]$	true if and only if underneath a RRA left foot
$\psi_b^{\Diamond}$	permitted specification for next terminal of type b

for each *deontic rule with anchor* $a \gg A > X < B \ll b$ *and permitted deontic function* $\Diamond$ **do**
  `/* Initialize Memo with RRA */`
  $\text{memo}[\text{row}, \text{col}] \leftarrow \text{LF}^{\text{RRA}}[\text{row} + \text{col}, \text{row}, X \rightarrow A\ B]$
  `/* Propagate Reachability */`
  for $\text{row} \leftarrow (n-1)$ **to** 0 **do**
    $\text{memo}[\text{row}, :]$ `|=` $\text{memo}[\text{row} + 1, :]$ `// vertical` ($\downarrow$)
    $\text{memo}[\text{row}, 1:]$ `|=` $\text{memo}[\text{row} + 1, :\text{-}1]$ `// diagonal` ($\searrow$)
  `/* Collect Active Antecedents */`
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    if $\text{type}(t_i) = a$ ***and*** $\text{memo}[0, i]$ **then**
      $\psi_b^{\Diamond} \leftarrow \psi_b^{\Diamond} \cup \Diamond(t_i)$
`/*` $\psi_b^{\Diamond}$ `is sent to enforcer */`

F Additional Experiments and Results

Figure 8 breaks down the success rate by the number of people in the puzzle, which is the measure of puzzle complexity built into the dataset by Xie et al. (2025).

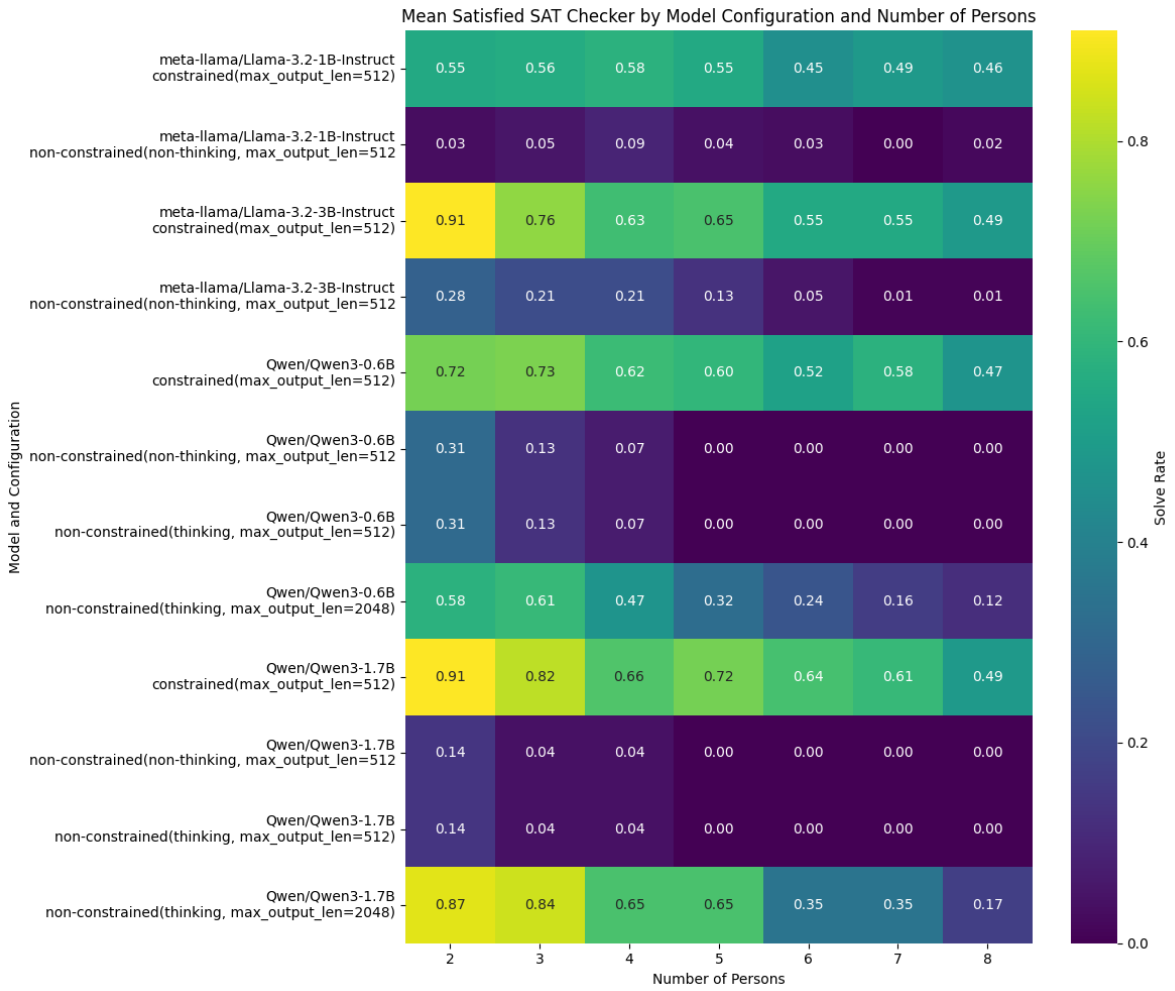

Figure 8: Knight and Knave test: Constrained models outperform unconstrained models across all puzzle sizes.