Finite-Choice Logic Programming

CHRIS MARTENS, Northeastern University, USA
ROBERT J. SIMMONS[*], Unaffiliated, USA
MICHAEL ARNTZENIUS[†], Unaffiliated, USA

Logic programming, as exemplified by datalog, defines the meaning of a program as its unique smallest model: the deductive closure of its inference rules. However, many problems call for an enumeration of models that vary along some set of choices while maintaining structural and logical constraints—there is no single canonical model. The notion of *stable models* for logic programs with negation has successfully captured programmer intuition about the set of valid solutions for such problems, giving rise to a family of programming languages and associated solvers known as answer set programming. Unfortunately, the definition of a stable model is frustratingly indirect, especially in the presence of rules containing free variables.

We propose a new formalism, *finite-choice logic programming*, that uses choice, not negation, to admit multiple solutions. Finite-choice logic programming contains all the expressive power of the stable model semantics, gives meaning to a new and useful class of programs, and enjoys a least-fixed-point interpretation over a novel domain. We present an algorithm for exploring the solution space and prove it correct with respect to our semantics. Our implementation, the Dusa logic programming language, has performance that compares favorably with state-of-the-art answer set solvers and exhibits more predictable scaling with problem size.

CCS Concepts: • **Computing methodologies → Logic programming and answer set programming**; • **Theory of computation → Constraint and logic programming**; **Denotational semantics**; • **Software and its engineering → Constraint and logic languages**.

Additional Key Words and Phrases: Datalog, answer set programming, possibility spaces

ACM Reference Format:
Chris Martens, Robert J. Simmons, and Michael Arntzenius. 2025. Finite-Choice Logic Programming. *Proc. ACM Program. Lang.* 9, POPL, Article 13 (January 2025), 29 pages. https://doi.org/10.1145/3704849

1 Introduction

Many important problem domains involve generating varied data according to structural and logical constraints. Examples include property-based random testing for typed functional programs [Claessen and Hughes 2011; Goldstein et al. 2023; Goldstein and Pierce 2022; Lampropoulos et al. 2017; Paraskevopoulou et al. 2022; Seidel et al. 2015], procedural content generation in games [Dormans and Bakkes 2011; Shaker et al. 2016; Short and Adams 2017], and software configuration [Czarnecki et al. 2004, 2002; Le et al. 2011]. To solve these problems, we write *generative programs,* characterized by *choice points* that create multiple possible outcomes and *constraints* that eliminate undesirable outcomes. From a declarative perspective, these generative programs

[*]The first and second authors contributed equally to the design of finite-choice logic programming and Dusa. The second author performed the implementation and analysis (Section 7).
[†]The first and third authors contributed equally to the fixed-point semantics in Sections 4 and 5.

Authors' Contact Information: Chris Martens, c.martens@northeastern.edu, Northeastern University, Boston, Massachusetts, USA; Robert J. Simmons, robsimmons@gmail.com, Unaffiliated, Boston, Massachusetts, USA; Michael Arntzenius, Unaffiliated, Hamilton Township, New Jersey, USA.

ACM 2475-1421/2025/1-ART13
https://doi.org/10.1145/3704849

characterize *possibility spaces*, and the meaning of a program is the entire space, which may be considered independently of particular methods for sampling individual solutions.

Using logic to describe possibility spaces has a long history, and its expressive power has been harnessed by established tools such as satisfiability solvers and logic programming languages. However, different logic-based tools disagree in important ways about how logical statements should be interpreted and thus how possibility spaces should be expressed.

Consider the problem of procedurally generating maps for a virtual world of connected regions where every region has one of three terrain types—*mountain*, *ocean*, or *forest*—and where oceans never directly adjoin mountains. We can model the possibility space of terrain maps as a Boolean satisfiability (SAT) problem by saying that the truth of a proposition $mountain(r)$ means that a region r has mountainous terrain (and so on):

$$(mountain(r) \lor forest(r) \lor ocean(r)) \leftrightarrow region(r) \tag{1}$$

$$(\neg mountain(r) \land \neg forest(r)) \lor (\neg mountain(r) \land \neg ocean(r)) \lor (\neg forest(r) \land \neg ocean(r)) \tag{2}$$

$$(adjacent(r_1, r_2) \land ocean(r_1)) \rightarrow (ocean(r_2) \lor forest(r_2)) \tag{3}$$

The conjunction of these three propositions neatly captures the desired possibility space: when augmented with a finite universe of possible regions and a collection of input literals characterizing the *adjacent* relation, a SAT solver would return valid terrain-to-region assignments.

Now suppose we additionally wish to enumerate all regions reachable from a given starting region by traversing only forests, perhaps with the goal of checking whether a player's initial position can reach both mountains and ocean. The fact that a region r is reachable from a starting region is naturally characterized with a recursively-defined proposition $reach(r)$ as follows:

$$reach(r) \leftarrow start(r) \tag{4}$$

$$reach(r_2) \leftarrow reach(r_1), forest(r_1), adjacent(r_1, r_2) \tag{5}$$

In this instance, logic programming gives the desired interpretation: from a rule $p \leftarrow q_1 \ldots q_n$ (where p and q_i are all positive assertions) and a database containing $q_1 \ldots q_n$, we deduce p. Such a program has a single canonical model, the smallest set of assertions closed under the given implications; any other assertion is deemed false [Van Gelder et al. 1991].

Unfortunately, these two approaches don't play well together. On one hand, logic programming lacks a notion of *choice point,* because propositions are interpreted as rules: not merely justifications for what *may* appear in the model, but assertions about what *must* appear in the model, so the first program has no direct analog in, say, datalog. On the other hand, defining reachability is notoriously difficult for SAT solvers: interpreting the reachability program as a logical formula using Clark's completion [1978], a SAT solver would validate solutions with spurious *reach* facts that have no justification imparted by the programmer.

The programming model we desire seems to alternate between two modes of operation: on the one hand, making *mutually exclusive choices* that multiply our possibility space and rule out certain combinations with other choices; and on the other hand, computing the *deductive consequences* of those choices. Unifying these perspectives is precisely what we achieve in this paper with finite-choice logic programming, a new approach to logic programming in which the generative constraints in formulas 1-3 can be expressed as follows:

$$terrain(r) \ \underline{\textsc{is}} \ \{\text{mountain}, \text{forest}, \text{ocean}\} \leftarrow region(r) \tag{6}$$

$$terrain(r_1) \ \underline{\textsc{is}} \ \{\text{forest}, \text{ocean}\} \leftarrow adjacent(r_1, r_2), terrain(r_2) \ \underline{\textsc{is}} \ \text{ocean} \tag{7}$$

The $\underline{\textsc{is}}$ syntax signals a *functional dependency* from the region r to its terrain type, and the rules represent mutually exclusive choices closed under deduction: a region must have one of three terrain types, and a region adjacent to the ocean must contain oceans or forests. In finite-choice logic programming, such rules combine naturally with rules like formulas 4 and 5; we would rewrite the latter rule as $\big(\mathit{reach}(r_2) \leftarrow \mathit{reach}(r_1), \mathit{terrain}(r_1)\ \underline{\text{IS}}\ \text{forest}, \mathit{adjacent}(r_1, r_2)\big)$.

1.1 Answer Set Programming and Its Discontents

Historically, the most successful approach for logic programming with mutually exclusive choices has been the *stable model semantics* introduced by Gelfond and Lifschitz [1988]. Stable models combine the Boolean-satisfiability intuitions behind Clark's completion with a rejection of circular justification, and provide the foundation for *answer set programming (ASP)*. Systems for computing the stable models of answer set programs have fruitfully co-evolved with the advancements in Boolean satisfiability solving, leading to sophisticated heuristics that make many problems fast in practice [Gebser et al. 2019]. ASP has seen considerable and ongoing success in procedural content generation [Dabral and Martens 2020; Neufeld et al. 2015; Smith and Bryson 2014; Smith et al. 2013; Smith and Mateas 2011; Summerville et al. 2018], analysis and scenario generation for legal regulations, policies, and contracts [Dabral et al. 2023; Lim et al. 2022], spatial reasoning for built structures [Li et al. 2020], and distributed system reasoning [Alvaro et al. 2011].

In a modern answer set programming language like Clingo [Gebser et al. 2011], the map-generation critera expressed in formulas 1-3 can be expressed concisely as follows:

$$1\,\{\ \mathit{mountain}(r);\ \mathit{forest}(r);\ \mathit{ocean}(r)\ \}\,1 \leftarrow \mathit{region}(r) \tag{8}$$

$$\leftarrow \mathit{adjacent}(r_1, r_2), \mathit{ocean}(r_1), \mathit{mountain}(r_2) \tag{9}$$

Formula 8 contains a "cardinality constraint" that insists on precisely one of the three terrain types holding for each region. Formula 9 is a "headless" rule, interpreted as a constraint forbidding the conjunction of all three premises from holding simultaneously.

However, the interpretation of answer set programs involves multiple levels of indirection that, in the authors' experience, make it challenging to reason about program meaning and performance. First, the stable model semantics that justifies answer set programming is defined only for logic programs without free variables. Answer set programming presents clients with an interface of rules containing free variables, such as formula 9 above; however, in a program with sixteen regions, the stable model semantics applies after expanding this rule into 256 variable-free rules, one for each assignment of regions to r_1 and r_2. This is reflected in essentially all implementations of answer set programming, which involve the interaction of a *solver* that only understands variable-free rules and a *grounder* that generates variable-free rules, usually incorporating heuristics to minimize the number of rules the solver must deal with [Kaufmann et al. 2016].

The second level of indirection is that even propositional answer set programs do not directly have a semantics: the definition of stable models involves a syntactic transformation of an answer set program into a logic program without negation (the *reduct*, see [Gelfond and Lifschitz 1988]). This transformation is defined with respect to a candidate model, and if the unique model of the reduct is the same as the candidate model, the candidate model is accepted as an actual model. This fixed-point-like definition is what the "stability" of stable models refers to.

The third level of indirection is that stable models are defined in terms of *negation* and not in terms of the higher-level constructs — choice, cardinality, and headless rules — that are foundational for essentially all modern applications of answer set programming. Higher-level constructs are either justified by translation to "pure" answer set programming with only negation (as in [Sacca and Zaniolo 1990]) or by appeal to a nonstandard definition of stable models (as in [Simons et al. 2002]).

1.2 A Constructive Semantics Based on Mutually-Exclusive Choices

In this paper we develop finite-choice logic programming as a logic programming language with *choice,* not negation, as primitive. We present and connect two semantics for finite-choice logic programming: a set-based semantics that describes the incremental construction of solutions (Section 2) and an interpretation of programs as the least fixed point of a monotonic immediate consequence operator over the domain of possibility spaces (Section 5). We demonstrate that finite-choice logic programming subsumes both datalog (Section 2.2) and answer set programming (Section 2.3), and present new examples of idiomatic finite-choice logic programs (Section 3). We present an algorithm for nondeterministic enumeration of solutions (Section 6) and describe an implementation (Section 7) that outperforms state-of-the-art answer set programming engines on a variety of examples, in part because our implementation avoids the "grounding bottleneck" encountered by ASP solvers that take a ground-then-solve approach to program execution.

2 Defining Finite-Choice Logic Programming

At the core of finite-choice logic programming are facts, which take the form $p(\bar{t})$ $\underline{\text{IS}}$ v, where $p(\bar{t})$ is the attribute and v is the unique value assigned to that attribute. A set of facts (which we'll call a *database*) must map each attribute to at most one value. Concretely, a fact like $\mathit{terrain}(\mathsf{home})$ $\underline{\text{IS}}$ forest indicates that the region named home contains forest terrain, and cannot contain any other terrain. This is called a *functional dependency*: the predicate $\mathit{terrain}$ is a partial function from region names to terrain types.

Definition 2.1 (Terms). As common in logic programming settings, terms are Herbrand structures, either variables $x, y, z, \ldots$ or uninterpreted functions $\mathsf{f}(t_1, \ldots, t_n)$ where the arguments t_i are terms. Constants are uninterpreted functions with no arguments, and as usual we'll leave the parentheses off and just write b or c instead of $\mathsf{b}()$ or $\mathsf{c}()$. We'll often abbreviate sequences of terms $t_1, \ldots, t_n$ as $\bar{t}$ when the indices aren't important.

Definition 2.2 (Facts). A fact has the form $p(t_1, \ldots, t_n)$ $\underline{\text{IS}}$ v, where p is a *predicate* and the t_i and v are variable-free (i.e. *ground*) terms. We call $p(t_1, \ldots, t_n)$ the fact's *attribute* and call the term v the fact's *value.* We will sometimes use a to stand in for a variable-free attributes $p(\bar{t})$.

Finite-choice logic programming admits specification of possibility spaces through the interplay of two kinds of rules:

Definition 2.3 (Rules). Rules $H \leftarrow F$ have one of two forms, *open* and *closed.*

$$p(\bar{t})\ \underline{\text{IS?}}\ v \leftarrow F \qquad \text{(open form rule)}$$

$$p(\bar{t})\ \underline{\text{IS}}\ \{v_1, \ldots, v_m\} \leftarrow F \qquad \text{(closed form rule, } m \geq 1\text{)}$$

In both cases, F is a conjunction of *premises* of the form $p(\bar{t})$ $\underline{\text{IS}}$ v. The rule's *conclusion* (or *head*) H is the part to the left of the $\leftarrow$ symbol. Both the head and the premises may contain variables, but every variable in the head must appear in a premise.

For intuition: a *closed* conclusion, such as $\mathit{terrain}(\mathsf{port})$ $\underline{\text{IS}}$ $\{\mathsf{ocean}, \mathsf{forest}\}$, *requires* that the region named port be either ocean or forest. An *open* conclusion, such as $\mathit{terrain}(\mathsf{goal})$ $\underline{\text{IS?}}$ meadow, requires that the attribute $\mathit{terrain}(\mathsf{goal})$ takes some value in the solution, and *permits* that value to be meadow. We make the semantics of these rule forms precise in the next section.

Definition 2.4 (Programs). A program P is a finite set of rules.

Definition 2.5 (Substitutions). A substitution σ is a total function from variables to ground terms. Applying a substitution to a term (σt) or a formula (σF) replaces all variables x in the term or formula with the term $\sigma(x)$.

2.1 Fact-Set Semantics

In this section, we present the meaning of finite-choice logic programs by describing their construction according to a nondeterminstic semantics.

Definition 2.6 (Databases). A *database* D is a set of variable-free facts $p(\bar{t})\ \underline{\text{IS}}\ v$ that is *consistent*, meaning that each attribute $p(\bar{t})$ maps to at most one value v: if $p(\bar{t})\ \underline{\text{IS}}\ v \in D$ and $p(\bar{t})\ \underline{\text{IS}}\ v' \in D$, then $v = v'$.

Definition 2.7 (Satisfaction). We say that a substitution σ *satisfies* F *in the database* D when, for each $p(\bar{t})\ \underline{\text{IS}}\ v$ in F, the fact $p(\sigma\bar{t})\ \underline{\text{IS}}\ \sigma v$ is present in D.

Definition 2.8 (Fact-set evolution). The relation $D \Rightarrow_P S$ relates a database (a consistent set of facts) to a set of databases (a set of consistent sets of facts):

- If P contains the closed-form rule $p(\bar{t})\ \underline{\text{IS}}\ \{v_1, \ldots, v_m\} \leftarrow F$ and σ satisfies F in D, then $D \Rightarrow_P S$, where S is the set of all $D \cup \{p(\sigma\bar{t})\ \underline{\text{IS}}\ \sigma v_i\}$ for $1 \le i \le m$ where the result of the union is consistent (and therefore a database).
- If P contains the open-form rule $p(t_1, \ldots, t_n)\ \underline{\text{IS?}}\ v \leftarrow F$ and σ satisfies F in D, then $D \Rightarrow_P S$, where S contains one or two elements. S always contains D itself, and if $D \cup \{p(\sigma\bar{t})\ \underline{\text{IS}}\ \sigma v\}$ is a consistent set of facts then S contains $D \cup \{p(\sigma\bar{t})\ \underline{\text{IS}}\ \sigma v\}$ as well.

To avoid a corner case with empty programs, we'll also say that $D \Rightarrow_P \{D\}$ always.

Definition 2.9 (Steps). We say that the program P allows D to *step* to D' if $D \Rightarrow_P S$ and $D' \in S$. Steps gives rise to *step sequences*: $D_1 \ldots D_k$ is a step sequence for P if $k \in \mathbb{N}$ (that is, if the sequence is finite) and if, for each $i > 1$, the program P allows D_{i-1} to step to D_i.

Definition 2.10 (Saturation). A database D is *saturated under a program* P if its *only possible evolution* under the $\Rightarrow_P$ relation is the singleton set containing itself. In other words, D is saturated under P if, for all S such that $D \Rightarrow_P S$, it is the case that $S = \{D\}$.

Definition 2.11 (Solutions). A *solution* to the program P is a saturated database D where $\emptyset \ldots D$ is a step sequence for P.

The implications of this definition of fact-set evolution are a bit subtle. In general, for a given program P and database D, there may be many S such that $D \Rightarrow_P S$, as many as there are pairs of substitutions σ and rules $H \leftarrow F$ such that σ satisfies F in P. This means that each step in a step sequence resolves two levels of nondeterminism: to take a step from D to D', first it is necessary to pick one of the possibly many S such that $D \Rightarrow_P S$, and then it is necessary to pick a database D' from that set S.

2.2 Simulating Datalog

A datalog program without negation is a set of rules (Horn clauses) where all variables in the conclusion appear somewhere in a premise.

$$p(\bar{t}) \leftarrow p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}) \qquad \text{(datalog rule)}$$

This is a generic use of "datalog," as often people take "Datalog" to specifically refer to "function-free" logic programs where term constants have no arguments, a condition sufficient to ensure that every program has a finite model. We follow many theoretical developments and practical implementations of datalog in ignoring the function-free requirement.

Finite-choice logic programming can simulate a datalog proposition $p(\bar{t})$ as a fact of the form $p(\bar{t})\ \underline{\text{IS}}\ \text{unit}$, where unit is a newly introduced constant. This is consistent with the traditional interpretation of datalog so long as the predicate p only appears in premises of the form $p(\bar{t})\ \underline{\text{IS}}\ \text{unit}$

$$p \; \underline{\text{IS?}} \; \text{ff} \leftarrow \tag{10}$$
$$q \; \underline{\text{IS?}} \; \text{ff} \leftarrow \tag{11}$$
$$p \; \underline{\text{IS}} \; \{\text{tt}\} \leftarrow q \; \underline{\text{IS}} \; \text{ff} \tag{12}$$
$$q \; \underline{\text{IS}} \; \{\text{tt}\} \leftarrow p \; \underline{\text{IS}} \; \text{ff} \tag{13}$$

Fig. 1. The finite-choice logic program corresponding to the two rules $p \leftarrow \neg q$ and $q \leftarrow \neg p$ in ASP.

or in conclusions of the form $p(\bar{t}) \; \underline{\text{IS}} \; \{\text{unit}\}$. This observation justifies our the use of value-free predicates in finite-choice logic programs, which we already saw with the use of $region(r)$ in rule 6 and $adjacent(r_1, r_2)$ in rule 7.

2.3 Simulating Answer Set Programming

Finite-choice logic programming can use the interplay of open and closed rules to obtain all the expressiveness of stable models without any reference to logical negation. An answer set program containing the two rules $p \leftarrow \neg q$ and $q \leftarrow \neg p$ corresponds to the finite-choice logic program in Figure 1. The open rules 10 and 11 unconditionally *permit* p or q to have the "false" value ff, and the closed rules 12 and 13 ensure that the assignment of ff to either p or q will force the other attribute to take the "true" value tt. (This asymmetry between the handling of truth and falsehood reflects their asymmetric treatment in answer set programming.) The two solutions for this program are $\{p \; \underline{\text{IS}} \; \text{tt}, q \; \underline{\text{IS}} \; \text{ff}\}$ and $\{p \; \underline{\text{IS}} \; \text{ff}, q \; \underline{\text{IS}} \; \text{tt}\}$, which correspond to the two solutions that answer set programming assigns to the source program.

Answer set programming is usually defined in terms of variable-free rules that have both non-negated premises p_i and negated premises $\neg q_i$.

$$p \leftarrow p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m \tag{ASP rule}$$

A stable model of an answer set program takes the form of a set X of variable-free propositions: any proposition in X is treated as true, and any proposition not in X is treated as false. (For the full definition of stable models, see Gelfond and Lifschitz [1988] or Appendix A.)

THEOREM 2.12. *Let P be a finite collection of ASP rules. There exists a translation of P to a finite-choice logic program $\langle P \rangle$ such that the following hold:*

- *For all stable models X of P, there is a solution D to $\langle P \rangle$ such that $X = \{p \mid p \; \underline{\text{IS}} \; \text{tt} \in D\}$.*
- *For all solutions D of $\langle P \rangle$, the set $\{p \mid p \; \underline{\text{IS}} \; \text{tt} \in D\}$ is a stable model of P.*

The proof of Theorem 2.12 is available in supplemental materials (Appendix A).

2.4 Example Program Execution

Let P be the four rule program from Figure 1. We demonstrate the fact-set semantics step-by-step, starting from the database $D_0 = \varnothing$. Two rules have satisfied premises and thus generate evolutions:

Rules:	**Evolutions:**
$p \; \underline{\text{IS?}} \; \text{ff} \leftarrow$	$\varnothing \Rightarrow_P \{ \varnothing, \; \{p \; \underline{\text{IS}} \; \text{ff}\} \} = S_1$
$q \; \underline{\text{IS?}} \; \text{ff} \leftarrow$	$\varnothing \Rightarrow_P \{ \varnothing, \; \{q \; \underline{\text{IS}} \; \text{ff}\} \} = S_2$
$p \; \underline{\text{IS}} \; \{\text{tt}\} \leftarrow q \; \underline{\text{IS}} \; \text{ff}$	*rule does not apply*
$q \; \underline{\text{IS}} \; \{\text{tt}\} \leftarrow p \; \underline{\text{IS}} \; \text{ff}$	*rule does not apply*

There are three sets S such that $\varnothing \Rightarrow_P S$: by rule 10 we have $\varnothing \Rightarrow_P S_1$, by rule 11 we have $\varnothing \Rightarrow_P S_2$, and by the trivial evolution $\varnothing \Rightarrow_P \{\varnothing\}$.

It requires only a slight misreading of the definitions to incorrectly conclude that $\varnothing$ is saturated: after all, in all cases that $\varnothing \Rightarrow_P S$, it is the case that $\varnothing \in S$. However, $\varnothing$ is *not* saturated by Definition 2.10, because $\varnothing$ can step to other databases as well.

If we choose to step from $\varnothing$ to $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$, three rules apply:

$$\begin{array}{rl}
p\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\}\ \} = S_3 \\
q\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\},\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}\ \} = S_4 \\
p\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow q\ \underline{\mathrm{IS}}\ \mathrm{ff} & \qquad\qquad \textit{rule does not apply} \\
q\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow p\ \underline{\mathrm{IS}}\ \mathrm{ff} & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}\ \} = S_5
\end{array}$$

As before, there are three possible evolutions and one of them is trivial. If we step to $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$ via S_4, we will find ourselves in trouble:

$$\begin{array}{rl}
p\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}\ \} \\
q\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}\ \} \\
p\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow q\ \underline{\mathrm{IS}}\ \mathrm{ff} & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \varnothing \\
q\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow p\ \underline{\mathrm{IS}}\ \mathrm{ff} & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \varnothing
\end{array}$$

This is yet another case where a slight misreading of the definitions could lead one to incorrectly conclude that $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$ is saturated: after all, we've just demonstrated that it is a database that can only step to itself. However, $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$ is not saturated by Definition 2.10: if we look at the last rule, it would seem to require that the program derive $q\ \underline{\mathrm{IS}}\ \mathrm{tt}$, conflicting with the existing fact $q\ \underline{\mathrm{IS}}\ \mathrm{ff}$. This conflict means that $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\} \Rightarrow_P \varnothing$, so the the database is not saturated by Definition 2.10. The database also cannot take a step to any other database aside from itself, and so this represents a failed search: no series of steps will lead to a solution. Note that this case demonstrates why we cannot define a saturated database as "a database that can only step to itself." Such a definition would—undesirably—admit databases that evolve to the empty set.

If we back up and instead let $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$ step to $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}$ via S_5, we can observe that the only way the database can evolve is to the singleton set containing itself, which is what Definition 2.10 required:

$$\begin{array}{rl}
p\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}\ \} \\
q\ \underline{\mathrm{IS?}}\ \mathrm{ff} \leftarrow & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}\ \} \\
p\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow q\ \underline{\mathrm{IS}}\ \mathrm{ff} & \qquad\qquad \textit{rule does not apply} \\
q\ \underline{\mathrm{IS}}\ \{\mathrm{tt}\} \leftarrow p\ \underline{\mathrm{IS}}\ \mathrm{ff} & \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\} \Rightarrow_P \{\ \{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}\ \}
\end{array}$$

Because $\{p\ \underline{\mathrm{IS}}\ \mathrm{ff}, q\ \underline{\mathrm{IS}}\ \mathrm{tt}\}$ is saturated, it is a solution. Symmetric reasoning applies to see that $\{p\ \underline{\mathrm{IS}}\ \mathrm{tt}, q\ \underline{\mathrm{IS}}\ \mathrm{ff}\}$ is a solution. By inspection, there are no solutions where both p and q are assigned the same value.

3 Finite-Choice Logic Programming by Example

This section presents examples to demonstrate common idioms that arise naturally in writing and reasoning about finite-choice logic programs.

$$\mathit{edge}(x, y) \leftarrow \mathit{edge}(y, x) \tag{14}$$
$$\mathit{root}\ \underline{\text{IS?}}\ x \leftarrow \mathit{edge}(x, y) \tag{15}$$
$$\mathit{parent}(x)\ \underline{\text{IS}}\ \{x\} \leftarrow \mathit{root}\ \underline{\text{IS}}\ x \tag{16}$$
$$\mathit{parent}(y)\ \underline{\text{IS?}}\ x \leftarrow \mathit{edge}(x, y), \mathit{parent}(x)\ \underline{\text{IS}}\ z \tag{17}$$

Fig. 2. Calculating a spanning tree over an undirected graph.

$$\mathit{edge}(x, y) \leftarrow \mathit{edge}(y, x) \tag{18}$$
$$\mathit{representative}(x)\ \underline{\text{IS?}}\ x \leftarrow \mathit{node}(x) \tag{19}$$
$$\mathit{representative}(y)\ \underline{\text{IS}}\ \{z\} \leftarrow \mathit{edge}(x, y), \mathit{representative}(x)\ \underline{\text{IS}}\ z \tag{20}$$

Fig. 3. Appointing a canonical representative for each connected component in an undirected graph.

3.1 Spanning Tree Creation

Seeded with an *edge* relation, the finite-choice logic program in Figure 2 will pick an arbitrary node and construct a spanning tree rooted at that node. The structure of this program is such that it's not possible to make forward progress that indirectly leads to conflicts: rule 15 can only apply once in a series of deductions, and rules 16 and 17 cannot fire at all until some root is chosen. A node can only be added to the tree once, with a parent that already exists in the tree, so this is effectively a declarative description of Prim's algorithm without weights.

The creation of an arbitrary spanning tree for an undirected graph is a common first benchmark for datalog extensions that admit multiple solutions. Most previous work makes a selection greedily, either discarding any future contradictory selections [Giannotti et al. 2001; Greco and Zaniolo 2001; Hu et al. 2021; Krishnamurthy and Naqvi 1988], or else avoiding contradictory deductions by consuming linear resources [Simmons and Pfenning 2008].

3.2 Appointing Canonical Representatives

When we want to check whether two nodes in an undirected graph are in the same connected component, one option is to compute the transitive closure of the *edge* relation. However, in a sparse graph, that can require computing $O(n^2)$ facts for a graph with n edges.

An alternative is to appoint an arbitrary member of each connected component as the canonical representative of that connected component: then, two nodes are in the same connected component if and only if they have the same canonical representative. This is the purpose of the program in Figure 3. In principle, it's quite possible for this program to get stuck in dead ends: if nodes a and b are connected by an edge, then rule 19 could appoint both nodes as a canonical representative, and rule 16 would then prevent any extension of that database from being a solution. In the greedy-choice languages mentioned in the previous section, this would be a problem for correctness: incorrectly firing an analogue of rule 19 would mean that a final database might contain two canonical representatives in a connected component. In finite-choice logic programming, because closed rules can lead to the outright rejection of a database, this is merely a problem of efficiency: we would like to avoid going down these dead ends.

In finite-choice logic programs like this one, we can reason about avoiding certain dead ends by assuming a mode of execution that we call **deduce, then choose**. The deduce-then-choose strategy dictates that, when picking the next step in a step sequence, we will always choose a non-trivial

$$
\begin{array}{rlr}
\mathit{visit}(\mathsf{z}) \leftarrow & \mathit{visit}(\mathsf{z})\ \underline{\text{is}}\ \{\mathsf{tt}\} \leftarrow & (21) \\
\mathit{visit}(\mathsf{s}(n)) \leftarrow \mathit{more}(n) & \mathit{visit}(\mathsf{s}(n))\ \underline{\text{is}}\ \{\mathsf{tt}\} \leftarrow \mathit{more}(n)\ \underline{\text{is}}\ \mathsf{tt} & (22) \\
 & \mathit{more}(n)\ \underline{\text{is?}}\ \mathsf{ff} \leftarrow \mathit{visit}(n)\ \underline{\text{is}}\ \mathsf{tt} & (23) \\
\mathit{stop}(n) \leftarrow \mathit{visit}(n), \neg\mathit{more}(n) & \mathit{stop}(n)\ \underline{\text{is}}\ \{\mathsf{tt}\} \leftarrow \mathit{visit}(n)\ \underline{\text{is}}\ \mathsf{tt}, \mathit{more}(n)\ \underline{\text{is}}\ \mathsf{ff} & (24) \\
 & \mathit{stop}(n)\ \underline{\text{is?}}\ \mathsf{ff} \leftarrow \mathit{visit}(n)\ \underline{\text{is}}\ \mathsf{tt} & (25) \\
\mathit{more}(n) \leftarrow \mathit{visit}(n), \neg\mathit{stop}(n) & \mathit{more}(n)\ \underline{\text{is}}\ \{\mathsf{tt}\} \leftarrow \mathit{visit}(n)\ \underline{\text{is}}\ \mathsf{tt}, \mathit{stop}(n)\ \underline{\text{is}}\ \mathsf{ff} & (26)
\end{array}
$$

Fig. 4. At left, an answer set program with no finite grounding. At right, the translation of this program to a finite-choice logic program.

evolution to a singleton ($D \Rightarrow_P \{D'\}$ with $D \neq D'$) over any evolution to a set containing two or more databases.

Endowed with the deduce-then-choose execution strategy, the program in Figure 3 will never make deductions that indirectly lead to conflicts. First, rule 18 will ensure that the edge relation is symmetric. Once that is complete, execution will be forced to choose some canonical representative using rule 19 in order to make forward progress. Once a representative is chosen, rule 20 will exhaustively assign that newly-appointed representative to every other node in the connected component. Only when it is done may rule 19 fire again for a node in another connected component.

3.3 Lazy Answer Set Programming

Theorem 2.12 only describes a correspondence between variable-free finite-choice logic programs and variable-free answer set programs: given that this is how answer set programming is usually formally defined, it was difficult to do otherwise. However, the program transformation underlying Theorem 2.12 applies to the non-ground answer set programs that are usually written down in practice, and all the translations we have attempted are faithful translations of the source answer set program. We confidently conjecture that the correctness of the translation extends to non-ground programs, but we leave the formal details for future work.

Translating non-ground answer set programs is interesting in part because of answer set programs like the one in Figure 4. That program has a well-defined set of solutions, but none of these solutions can be enumerated by mainstream answer set programming implementations because the solver invokes an initial grounding step that is forced to generate an infinite set of ground rules. The translation of this program as a finite-choice logic program, on the other hand, has a simple operational interpretation under the deduce-then-choose strategy: starting from zero, each simple round of deduction will *visit* a successively larger natural number, at which point a choice will be made to either *stop* at that number or to continue to visit *more* numbers.

This example suggests a connection between finite-choice logic programming and the strategy of using *lazy grounding* in answer set programming [Dal Palù et al. 2009], a strategy which similarly enables the evaluation of answer set programs with no finite grounding [Comploi-Taupe et al. 2023]. We will return to this point in Section 7.2.

3.4 Satisfiability

The previous examples all show how the evaluation of finite-choice logic programming can entirely avoid reaching databases that are not solutions. However, the full expressive power of finite-choice logic programming comes from the ability to represent problems where that avoidance is not possible. Since answer set programming generalizes boolean satisfiability, it is no surprise

$$p \ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \leftarrow \tag{27}$$
$$q \ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \leftarrow \tag{28}$$
$$r \ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \leftarrow \tag{29}$$
$$ok \ \underline{\text{IS}}\ \{\text{yes}\} \leftarrow \tag{30}$$
$$ok \ \underline{\text{IS}}\ \{\text{no}\} \leftarrow p \ \underline{\text{IS}}\ \text{ff}, q \ \underline{\text{IS}}\ \text{tt} \tag{31}$$
$$ok \ \underline{\text{IS}}\ \{\text{no}\} \leftarrow p \ \underline{\text{IS}}\ \text{tt}, q \ \underline{\text{IS}}\ \text{ff}, r \ \underline{\text{IS}}\ \text{ff} \tag{32}$$

Fig. 5. A finite-choice logic program representing the SAT instance $(p \vee \neg q) \wedge (\neg p \vee q \vee r)$.

that boolean satisfiability problems can be represented straightforwardly in finite-choice logic programming.

A Boolean satisfiability problem can be written as a conjunction of clauses where each clause is a disjunction of propositions p and negated propositions $\neg p$. We represent such problems by explicitly assigning each proposition to tt or ff by a closed rule, and adding a rule for each clause that causes a value conflict for the ok predicate if the clause's negation holds; see Figure 5 for an example. The deduce-then-choose execution strategy gives no advantages here: the only deduction is observing inconsistencies that result from already-selected choices.

4 Nondeterministic Immediate Consequences

The semantics given in Section 2 allow us to interpret the meaning of a finite-choice logic program as the set of the program's solutions (Definition 2.11). In some ways, though, this semantics leaves a lot to be desired. Consider the following program P:

$$p \ \underline{\text{IS}}\ \{\text{a}, \text{b}\} \leftarrow \tag{33}$$
$$p \ \underline{\text{IS}}\ \{\text{b}, \text{c}\} \leftarrow \tag{34}$$
$$q \ \underline{\text{IS?}}\ \text{ff} \leftarrow \tag{35}$$
$$q \ \underline{\text{IS}}\ \{\text{tt}\} \leftarrow p \ \underline{\text{IS}}\ x \tag{36}$$

Under the semantics in Section 2, there is a two-step sequence from $\varnothing$ to $\{p \ \underline{\text{IS}}\ \text{c}, q \ \underline{\text{IS}}\ \text{ff}\}$:

$$\varnothing \Rightarrow_P \{\ \{p \ \underline{\text{IS}}\ \text{b}\},\ \underline{\{p \ \underline{\text{IS}}\ \text{c}\}}\ \} \qquad \text{(by rule 34)}$$
$$\{p \ \underline{\text{IS}}\ \text{c}\} \Rightarrow_P \{\ \{p \ \underline{\text{IS}}\ \text{c}\},\ \underline{\{q \ \underline{\text{IS}}\ \text{ff}, p \ \underline{\text{IS}}\ \text{c}\}}\ \} \qquad \text{(by rule 35)}$$

The result is not a solution, cannot be extended to a solution, and also cannot step to any other database. The first step led us, irrevocably, to a dead end: the step sequence $\varnothing, \{p \ \underline{\text{IS}}\ \text{c}\}, \ldots$ can never be extended to reach a solution.

Some dead ends are unavoidable when conflicting assignments only occur down significant chains of deduction. That possibility is part of what gives finite-choice logic programming its expressive power! In the program above, though, the conflict is in some sense immediate: at each step, we have enough information to know that rule 33 and rule 34 both apply, and the overlap of these closed rules means that p can only be given the value b in any solution.

In this section, we will revisit the semantics of finite-choice logic programming to present a *immediate consequence* operator that captures global information about what is "immediately derivable" from a given program and database. This development will provide the basis for a least-fixed-point semantics of finite-choice logic programming (Section 5), as well as the foundation for our implementation of finite-choice logic programming (Section 6).

4.1 Bounded-Complete Posets

This development introduces several new concepts: *constraints*, *constraint databases*, and *choice sets*. All these are instances of the same semilattice-like structure: bounded-complete posets.

In Section 2, we define databases as sets with an auxiliary definition of consistency. Now we will generalize consistency to a notion of *compatibility:*

Definition 4.1 (Compatibility). If $\mathscr{D}$ is a set equipped with a partial order $\leq$, then a subset $X \subseteq \mathscr{D}$ is *compatible* when it has an upper bound, i.e. $\exists y \in \mathscr{D}.\ \forall x \in X.\ x \leq y$. We write $\| X$ to assert that X is compatible and $\not\| X$ for its negation. As a binary operator $x \parallel y = \|\{x, y\}$ and $x \not\| y = \not\|\{x, y\}$.

We will frequently use this basic fact about (in)compatibility:

Lemma 4.2. *Compatibility is anti-monotone and incompatibility is monotone: if $D \leq D'$ and $E \leq E'$, then $D' \parallel E' \implies D \parallel E$, and contrapositively $D \not\| E \implies D' \not\| E'$.*

Proof. $D' \parallel E'$ means D', E' have some upper bound D^*; if $D \leq D'$ and $E \leq E'$, then D^* is also an upper bound for D, E. □

Definition 4.3. A *bounded-complete poset* $(\mathscr{D}, \leq_{\mathscr{D}}, \bot_{\mathscr{D}}, \bigvee_{\mathscr{D}})$ is a poset with a least element where all compatible subsets have least upper bounds. In detail:

(1) $\leq_{\mathscr{D}} \subseteq \mathscr{D} \times \mathscr{D}$ is a partial order (a reflexive, transitive, antisymmetric relation).
(2) $\bot_{\mathscr{D}} \in \mathscr{D}$ is the least element: $\forall x \in \mathscr{D}.\ \bot \leq x$.
(3) $\bigvee_{\mathscr{D}} : \{X \subseteq \mathscr{D} : \| X\} \to \mathscr{D}$ finds the least upper bound of a compatible set of elements: $(\forall x \in X.\ x \leq z) \implies \bigvee_{\mathscr{D}} X \leq z$. As a binary operator, $x \vee_{\mathscr{D}} y = \bigvee_{\mathscr{D}}\{x, y\}$.

Example 4.4. Databases as presented in Section 2.1 are bounded-complete posets, where $D_1 \leq D_2$ is the subset relation and $\bot$ is the empty set $\varnothing$. The least upper bound operation is set union: if a set of databases are each individually subsets of some consistent set D of facts, then their union is a subset of D and so must also be a consistent set of facts.

4.2 Constraints and Constraint Databases

Definition 4.5. A *constraint*, $c \in \mathit{Constraint}$, is either $(\mathsf{just}\ t)$ for some ground term t or $(\mathsf{noneOf}\ X)$ for some set X of ground terms. Constraints form a bounded-complete poset as follows:

(1) $\leq_{\mathit{Constraint}}$ is defined by cases:

$$\begin{aligned} \mathsf{noneOf}\ X \leq \mathsf{noneOf}\ Y &\iff X \subseteq Y \\ \mathsf{noneOf}\ X \leq \mathsf{just}\ t &\iff t \notin X \\ \mathsf{just}\ t \leq \mathsf{just}\ t' &\iff t = t' \\ \mathsf{just}\ t \not\leq \mathsf{noneOf}\ X & \end{aligned}$$

(2) $\bot_{\mathit{Constraint}} = \mathsf{noneOf}\ \varnothing$.
(3) $\bigvee_{\mathit{Constraint}} C$ is defined by cases. Because the least upper bound is only defined for compatible sets, if we know $\mathsf{just}\ t \in C$ then $\mathsf{just}\ t$ is the least upper bound: any other upper bound must have the form $\mathsf{just}\ t'$ with $t = t'$. Otherwise, every $c_i \in C$ is of the form $\mathsf{noneOf}\ X_i$, and their least upper bound is $\mathsf{noneOf}\ (\bigcup_i X_i)$. By way of illustration, in the binary case:

$$\begin{aligned} \mathsf{noneOf}\ X \vee \mathsf{noneOf}\ Y &= \mathsf{noneOf}\ (X \cup Y) & \\ \mathsf{noneOf}\ X \vee \mathsf{just}\ t &= \mathsf{just}\ t & \text{if } t \notin X \\ \mathsf{just}\ t \vee \mathsf{noneOf}\ X &= \mathsf{just}\ t & \text{if } t \notin X \\ \mathsf{just}\ t \vee \mathsf{just}\ t' &= \mathsf{just}\ t & \text{if } t = t' \end{aligned}$$

If none of these cases apply, $c_1 \vee c_2$ is undefined because $c_1 \not\| c_2$.

Definition 4.6. A *constraint database, $D, E \in DB$*, is a map from ground attributes a to constraints $D[a]$. Constraint databases form a bounded-complete poset with structure inherited pointwise from *Constraint:*

(1) $D \leq_{DB} E \iff \forall a.\, D[a] \leq E[a]$.
(2) $\bot_{DB}$ is mapping that takes every attribute a to $\bot_{Constraint} = \mathsf{noneOf}\ \varnothing$.
(3) $\bigvee_{DB} S = a \mapsto \bigvee_{Constraint} \{D[a] : D \in S\}$, and so is defined whenever $\forall a.\ \Vert\{D[a] : D \in S\}$.

Definition 4.7. A constraint database D is *positive* if $D[a] = \mathsf{noneOf}\ X$ implies $X = \varnothing$.

Definition 4.8. A constraint database D is *finite* if $\{a : D[a] \neq \mathsf{noneOf}\ \varnothing\}$ is finite and all X such that $D[a] = \mathsf{noneOf}\ X$ are finite.

We will introduce a new notation to describe finite constraint databases: $(p \mapsto \mathsf{just}\ \mathsf{ff},\ q \mapsto \mathsf{noneOf}\ \{\mathsf{ff}\})$ represents the constraint database that takes p to $\mathsf{just}\ \mathsf{ff}$, takes q to $\mathsf{noneOf}\ \{\mathsf{ff}\}$, and takes every other attribute to $\bot_{Constraint} = \mathsf{noneOf}\ \varnothing$.

There is an obvious isomorphism between positive constraint databases and the databases-as-consistent-sets-of-facts as introduced in Definition 2.6. The consistent sets of facts $\varnothing$, $\{p\ \underline{\text{is}}\ \mathsf{tt},\ q\ \underline{\text{is}}\ \mathsf{ff}\}$, and $\{edge(\mathsf{a}, \mathsf{b})\ \underline{\text{is}}\ \mathsf{unit}\}$ correspond, respectively, to the constraint databases $\bot_{DB}$, $(p \mapsto \mathsf{just}\ \mathsf{tt},\ q \mapsto \mathsf{just}\ \mathsf{ff})$, and $(edge(\mathsf{a}, \mathsf{b}) \mapsto \mathsf{just}\ \mathsf{unit})$.

4.3 Why Constraint Databases Aren't Enough

The move from "sets of facts" to "functions from attributes to constraints" is an instance of a common pattern encountered in extensions of logic programming to non-Boolean values. In the bilattice-annotated logic programming setting, Fitting [1993] calls the analogue of a constraint database a *valuation*, and Komendantskaya and Seda [2009] call the analogue of a constraint database an *annotation Herbrand model.* In the weighted logic programming setting, Eisner [2023] maintains a similar map ω from items to weights.

To our knowledge, all of this related work proceeds to define an immediate consequence operator as a function from database-analogues to database-analogues, and the meaning of a program is given as the unique least fixed point of this operator. But in finite-choice logic programming, as in answer set programming, programs can have multiple incompatible solutions, and so the meaning of a program *cannot* be a singular least fixed point of a function from constraint databases to constraint databases. One way forward is to present an operator where solutions correspond to *any* fixed point, or correspond to any fixed point of the operator that satisfies some additional condition: this is essentially the approach used by Fitting [1993] to bound the set of stable models for an answer set program using bilattices. However, this formulation doesn't directly characterize the stable models: it is still necessary to carry out the Gelfond-Lifschitz program transformation as a post-hoc check to see if one of Fitting's bounded fixed points is a stable model. Our attempts to precisely characterize solutions for finite-choice logic programs as an arbitrary fixed point of some function from databases to databases encountered similar difficulties.

In this work, we take a different approach, which we believe is novel: in Section 5, the least-fixed-point interpretation of finite-choice logic programming is defined to be a *set of pairwise incompatible constraint databases*, which we call a *choice set.*

4.4 Choice Sets

Choice sets form not merely a bounded-complete poset, but a complete lattice that has all least upper bounds. We will establish this in two steps, first defining choice sets as a pointed partial order, and then defining least upper bounds.

Definition 4.9. A *choice set* $C \in \mathit{Choice}$, is a *pairwise-incompatible* set of constraint databases, meaning that $(\forall D, E \in C.\, D \parallel E \implies D = E)$. *Choice* is a pointed partial order:

(1) $C_1 \leq_{\mathit{Choice}} C_2 \iff (\forall D_2 \in C_2.\, \exists D_1 \in C_1.\, D_1 \leq D_2) \iff (\forall D_2 \in C_2.\, \exists! D_1 \in C_1.\, D_1 \leq D_2)$. Existence $\exists D_1$ implies *unique* existence $\exists! D_1$ by pairwise incompatibility: if $D_1, D_1' \in C_1$ are both $\leq D_2$ they are compatible and thus equal.
(2) $\bot_{\mathit{Choice}} = \{\bot_{\mathit{DB}}\}$, that is, the set containing one item, the constraint database that maps every attribute to noneOf $\varnothing$.
(3) $\top_{\mathit{Choice}} = \varnothing$.

In a constraint database D, each rule and rule-satisfying substitution in a finite-choice logic program will induce a choice set C_i, and deriving the immediate consequences of D involves a "parallel composition" of all these choices. The least upper bound $\bigvee_i C_i$ calculates this parallel composition. We will start with a finite example to build intuition:

Example 4.10. If we have $C_1 = \{D_a, D_b, D_c\}$ and $C_2 = \{D_x, D_y\}$, then $C_1 \vee C_2$ contains between zero and six elements: all of the least upper bounds out of $(D_a \vee D_x)$, $(D_b \vee D_x)$, $(D_c \vee D_x)$, $(D_a \vee D_y)$, $(D_b \vee D_y)$, and $(D_c \vee D_y)$ that are actually defined.

For the least upper bound of n choice sets, a database is in the least upper bound exactly when it is the least upper bound of n compatible databases, one drawn from each of the choice sets. Formally, we define the least upper bound of a collection $C_{i \in I}$ of choice sets indexed by some set I. Let $f : I \to \mathit{DB}$ be a function choosing one database from each choice set, so that $f(i) \in C_i$. If the set of databases thus chosen, $\mathrm{Im}(f) = \{f(i) : i \in I\}$, is compatible, we include its least upper bound $\bigvee_i f(i)$ in the resulting choice set $\bigvee_i C_i$. The set of chosen databases, $\mathrm{Im}(f) = \{f(i) : i \in I\}$, can be seen as a *candidate set* for the least upper bound. If $\mathrm{Im}(f)$ is compatible, the least upper bound of $\mathrm{Im}(f)$ exists and we include its least upper bound in the resulting choice set $\bigvee_i C_i$.

Definition 4.11 (Least upper bounds for Choice). Take any $\{C_i : i \in I\} \subseteq \mathit{Choice}$, and let $\prod_{i \in I} C_i$ be the set of all functions $f : I \to \mathit{DB}$ such that $f(i) \in C_i$. Then:

$$\bigvee_{i \in I} C_i = \left\{ \bigvee \mathrm{Im}(f) \;:\; f \in \prod_{i \in I} C_i,\; \|\mathrm{Im}(f) \right\}$$

It is not entirely trivial to show that $\bigvee_i C_i$ is a least upper bound in *Choice:* the proof is available in the supplemental material (Appendix B). The main subtleties are ensuring that we avoid combining incompatible databases and ensuring that the resulting set remains pairwise incompatible.

The partial order $\leq_{\mathit{Choice}}$, unlike the partial orders on *Constraint* and *DB*, has a greatest element ($\varnothing$), and so any collection of choice sets has an upper bound. Therefore, *Choice* is a complete lattice, not merely a bounded-complete poset.

4.5 Immediate Consequence

Almost everything is now in place define an immediate consequence operator $\tau_P : \mathit{DB} \to \mathit{Choice}$. We only need to replay the definition of satisfaction from Section 2.1 in terms of constraint databases and choice sets, and then we can define the immediate consequence as the least upper bound of the consequence of every rule that can fire.

Definition 4.12 (Satisfaction). We say that a substitution σ *satisfies F in the constraint database D* when, for each premise $p(\bar{t})$ $\underline{\text{is}}$ v in F, we have $(\text{just } \sigma v) \leq D[p(\sigma\bar{t})]$.

Definition 4.13. A ground rule conclusion H defines a element of *Choice*, which we write as $\langle H \rangle$, in the following way:

- $\langle p(\bar{t})\ \underline{\text{IS?}}\ v\rangle = \{\ (p(\bar{t}) \mapsto \mathsf{just}\ v),\ \ (p(\bar{t}) \mapsto \mathsf{noneOf}\ \{v\})\ \}$
- $\langle p(\bar{t})\ \underline{\text{IS}}\ \{v_1, \ldots, v_n\}\rangle = \{\ (p(\bar{t}) \mapsto \mathsf{just}\ v_1),\ \ldots,\ (p(\bar{t}) \mapsto \mathsf{just}\ v_n)\ \}$

Definition 4.14 (Immediate consequence). The immediate consequence operator $\tau_P : DB \longrightarrow Choice$ is the least upper bound of every head of a rule with satisfied premises:

$$\tau_P(D) = \{D\} \vee \left(\bigvee \{\langle \sigma H\rangle : (H \leftarrow F) \in P, \sigma F \leq D\}\right)$$

We forcibly ensure that $\{D\} \leq \tau_P(D)$ by making the result the least upper bound of $\{D\}$ itself and the least upper bound of all the satisfied rule heads. (We conjecture this is redundant for our subsequent developments, but it is also harmless.)

Example 4.15. Returning to the four rule program P at the beginning of this section (rules 33-36), these are all examples of how the immediate consequence operator for that program behaves on different inputs:

$$\begin{aligned}
\tau_P(\bot_{DB}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{ff}),\\
&\qquad (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{noneOf}\ \{\mathsf{ff}\})\ \}\\
\tau_P(p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{ff}) &= \varnothing\\
\tau_P(p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{noneOf}\ \{\mathsf{ff}\}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{tt})\ \}\\
\tau_P(q \mapsto \mathsf{just}\ \mathsf{ff}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{ff})\ \}\\
\tau_P(p \mapsto \mathsf{just}\ \mathsf{b}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{tt})\ \}\\
\tau_P(p \mapsto \mathsf{just}\ \mathsf{b}, q \mapsto \mathsf{just}\ \mathsf{tt}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{tt})\ \}\\
\tau_P(q \mapsto \mathsf{just}\ \mathsf{someOtherTerm}) &= \{\ (p \mapsto \mathsf{just}\ \mathsf{b},\ q \mapsto \mathsf{just}\ \mathsf{someOtherTerm})\ \}
\end{aligned}$$

The immediate consequence operator on $\bot_{DB}$ captures the fact that that, due to rules 33 and 34 in combination, p can only be assigned the value b. Any other step, like a step to $p\ \underline{\text{IS}}\ \mathsf{a}$ by rule 33, is a dead end. The immediate consequence operator does not, however, preclude the dead-end step from an empty database to a database where $q\ \underline{\text{IS}}\ \mathsf{ff}$, since that information is not immediately available: rule 36 only applies in a database where p has a definite value.

5 Fixed-Point Semantics

We have one notion of what a finite-choice logic program means: the meaning of a program P is the set of databases that are solutions to P according to Definition 2.11. So why do need to provide another explanation for what a program means?

In our view, there are three satisfying ways of explaining the foundations of a logic programming language. In no particular order:

- A logic program defines a set of propositions in a constructive proof theory, and the meaning of a program is given by the logical consequences of those propositions. The methodology of *uniform proofs* brings with it a built-in operational semantics in terms of proof search in a focused sequent calculus [Miller et al. 1991].
- A logic program defines a monotonic *immediate consequence* function, and the meaning of a logic program is the least fixed point of this function. This methodology brings with it a built-in operational semantics, because one can repeatedly apply the immediate consequence operation to a least element in hopes of reaching a fixed point [Van Gelder et al. 1991].
- A logic program defines a proposition in classical logic through the program completion of Clark [1978], and the meaning of a logic program is the set of satisfying models of that proposition. This interpretation admits spurious circular justifications: the program containing one rule $p \leftarrow p$ has the Clark completion $p \leftrightarrow p$, which admits both the expected

interpretation (p is false) as well as an undesirable interpretation (p is true). Some programs, however, have a unique *least* model — the smallest set of true propositions — which can be interpreted as the canonical solution.

A very nice property of datalog is that it admits all three justifications, and they all agree. Answer set programming provides a greatly desirable property — the ability to give a program multiple, mutually-exclusive solutions — but it does so at the cost of being able to give programs any of these foundationally satisfying semantics.

In this section, we will present the interpretation of a finite-choice logic program as a member of *Choice*, the least fixed point of a "lifted" immediate consequence operator that has choice sets as both its domain and range. Theorem 5.12 establishes that solutions (Definition 2.11) correspond to databases in this least fixed point that are positive (Definition 4.7) and finite (Definition 4.8).

Example 5.1. First, let's develop some intuition about what it means for a choice set to provide the interpretation of a finite-choice logic program. In this example only, allow yourself to squint and reinterpret consistent sets of facts as constraint databases: in this hazy light, the set of solutions for a given program looks a lot like a choice set.

The program with no rules corresponds to the choice set $\bot_{Choice} = \{\varnothing\}$.

The program with one rule ($p\ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \longleftarrow$) has two pairwise-incompatible solutions that form the choice set $\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \}$.

$$\bot_{Choice} \leq_{Choice} \{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \}$$

One way additional rules can create greater choice sets is by adding facts. A second rule ($q\ \underline{\text{IS}}\ \{\text{tt}\} \longleftarrow$) results in a program that still has two solutions.

$$\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \} \leq_{Choice} \{\ \{p\ \underline{\text{IS}}\ \text{tt}, q\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}, q\ \underline{\text{IS}}\ \text{tt}\}\ \}$$

If we instead added a more constrained second rule ($q\ \underline{\text{IS}}\ \{\text{tt}\} \leftarrow p\ \underline{\text{IS}}\ \text{tt}$), we would instead have these solutions:

$$\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \} \leq_{Choice} \{\ \{p\ \underline{\text{IS}}\ \text{tt}, q\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \}\ \}$$

Another way additional rules can create greater choice sets is by *removing solutions*. If we return to the program with just the rule ($p\ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \longleftarrow$) and add a second rule ($p\ \underline{\text{IS}}\ \{\text{tt}\} \leftarrow p\ \underline{\text{IS}}\ x$), the database where $p\ \underline{\text{IS}}\ \text{ff}$ is no longer a solution:

$$\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \} \leq_{Choice} \{\ \{p\ \underline{\text{IS}}\ \text{tt}\}\ \}$$

A choice set that is greater according to $\leq_{Choice}$ may alternatively contain *more* constraint databases, which would occur if the second rule was instead ($q\ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \leftarrow p\ \underline{\text{IS}}\ \text{ff}$):

$$\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \} \leq_{Choice} \{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}, q\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}, q\ \underline{\text{IS}}\ \text{ff}\}\ \}$$

Finally, if we have a fully contradictory program, such as one containing two rules ($p\ \underline{\text{IS}}\ \{\text{tt}, \text{ff}\} \longleftarrow$) and ($p\ \underline{\text{IS}}\ \{\text{meadow}\} \leftarrow p\ \underline{\text{IS}}\ x$), its interpretation is the set containing zero solutions. This is a valid choice set, and is in fact the greatest element of *Choice*.

$$\{\ \{p\ \underline{\text{IS}}\ \text{tt}\},\ \{p\ \underline{\text{IS}}\ \text{ff}\}\ \} \leq_{Choice} \varnothing = \top_{Choice}$$

Running through this example, one can observe a kind of monotonicity at work: adding rules to a program always results in the meaning of that larger program being a greater choice set according to the partial order $\leq_{Choice}$ from Definition 4.9.

5.1 Lifted Immediate Consequence

The typical move when defining the meaning of a forward-chaining logic programs is to interpret the program as the least fixed point of an immediate consequence function. Immediate consequence presented in Section 4.5 is a function $\tau_P : DB \longrightarrow Choice$, and because the domain and range are different we can't take the fixed point. Recalling the definition of saturation in Definition 2.10 gives us a limited (but important!) notion of fixed points that we call *models*:

Definition 5.2. A constraint database D is a *model* of the program P in either of these equivalent conditions:

$$D \in \tau_P(D) \iff \tau_P(D) = \{D\}$$

We have no hope of giving a least-fixed-point interpretation of finite-choice logic programs this way, as most interesting finite-choice logic programs do not have unique minimal models. Instead, we will "lift" τ_P to a function $\mathcal{T}_P : Choice \longrightarrow Choice:$

Definition 5.3. $\mathcal{T}_P(C) = \bigcup_{D \in C} \tau_P(D)$.

For Definition 5.3 to be a candidate for iterating to a fixed point, we must show that $\bigcup_{D \in C} \tau_P(D)$ is in *Choice*, i.e. that all the databases it contains are pairwise incompatible:

Lemma 5.4. *If C is pairwise incompatible, then so is $\bigcup_{D \in C} \tau_P(D)$. That is, if $E_1, E_2 \in \bigcup_{D \in C} \tau_P(D)$ are compatible, then they are equal.*

Proof. Consider some $D_1, D_2 \in C$ and $E_1 \in \tau_P(D_1)$ and $E_2 \in \tau_P(D_2)$. Suppose $E_1 \parallel E_2$. Since $D_1 \leq E_1$ and $D_2 \leq E_2$, by Lemma 4.2 we know $D_1 \parallel D_2$, thus $D_1 = D_2$ (since both are in C). And since $\tau_P(D_1) = \tau_P(D_2)$ is pairwise incompatible, $E_1 = E_2$. □

Now we would like to define the meaning of a finite-choice logic program as the unique least fixed point of $\mathcal{T}_P$. Because choice sets form a complete lattice, we can apply Knaster-Tarski [Tarski 1955] as long as $\mathcal{T}_P$ is monotone, which it is:

Lemma 5.5 (τ_P is monotone). *If $D_1 \leq_{DB} D_2$, then $\tau_P(D_1) \leq_{Choice} \tau_P(D_2)$.*

Proof. Because $\tau_P(D_1)$ and $\tau_P(D_2)$ are the least upper bound of the rule heads satisfied in D_1 and D_2 respectively, it suffices to show that if σ satisfies F in D_1, then σ satisfies F in D_2. This follows from Definition 4.12: for each premise $p(\bar{t})$ is v in F, we have just $\sigma v \leq D_1[p(\sigma\bar{t})] \leq D_2[p(\sigma\bar{t})]$. □

Lemma 5.6 ($\mathcal{T}_P$ is monotone). *If $C_1 \leq C_2$, then $\mathcal{T}_P(C_1) \leq \mathcal{T}_P(C_2)$.*

Proof. We wish to show $\bigcup_{D \in C} \tau_P(D) \leq_{Choice} \bigcup_{D' \in C'} \tau_P(D')$. So, fixing $D' \in C'$ and $E' \in \tau_P(D')$, we wish to find a $D \in C$ and $E \in \tau_P(D)$ with $E \leq E'$. Since $C \leq C'$, for $D' \in C'$ there exists a unique $D \in C$ with $D \leq D'$. By monotonicity of τ_P we have $\tau_P(D) \leq_{Choice} \tau_P(D')$. Thus for $E' \in \tau_P(D')$ we have a unique $E \in \tau_P(D)$ with $E \leq E'$. □

Theorem 5.7 (Least fixed points). *$\mathcal{T}_P$ has a least fixed point, written as* $\operatorname{lfp} \mathcal{T}_P$.

Proof. Because $\mathcal{T}_P$ is monotone and *Choice* is a complete lattice, by Tarski [1955], the set of fixed points of $\mathcal{T}_P$ forms a complete lattice. The least fixed point is the least element of this lattice, i.e. $\operatorname{lfp} \mathcal{T}_P = \bigwedge\{C : \mathcal{T}_P(C) \leq C\}$ (where $\bigwedge X = \bigvee\{C : \forall x \in X, C \leq x\}$). □

Corollary 5.8. *Every model E has a lower bound $D \in \operatorname{lfp} \mathcal{T}_P$ with $D \leq E$.*

Proof. Since $\{E\}$ is a fixed point of $\mathcal{T}_P$, we know $\operatorname{lfp} \mathcal{T}_P \leq_{Choice} \{E\}$, i.e. $\exists D \in \operatorname{lfp} \mathcal{T}_P.\ D \leq E$. □

Theorem 5.9 (Minimal models). $\operatorname{lfp} \mathcal{T}_P$ *contains exactly the minimal models of P, meaning models D such that any model $D' \leq D$ is equal to D.*

Proof. No model outside $\text{lfp}\,\mathcal{T}_P$ is minimal, because by Corollary 5.8 it has a lower bound in $\text{lfp}\,\mathcal{T}_P$. And every model $D \in \text{lfp}\,\mathcal{T}_P$ is minimal: given a model $E \leq D$, by Corollary 5.8, there is some $D' \in \text{lfp}\,\mathcal{T}_P$ with $D' \leq E \leq D$. But then $D' \parallel D$ and by pairwise incompatibility $D' = D = E$. □

5.2 Agreement of Step Semantics and Least-Fixed-Point Semantics

Theorem 5.7 establishes that it is reasonable to say that a finite-choice logic program has a canonical model, $\text{lfp}\,\mathcal{T}_P$. Defining the meaning of a program this way, instead of the step-by-step definition given in Section 2.1, has a number of advantages. However, we need to take care with stating the correspondence between our two ways of assigning meaning to a finite-choice logic program.

Example 5.10. The the one-rule program $(p \ \underline{\text{is?}}\ \mathsf{b} \longleftarrow)$ has one solution according to Definition 2.11, the set $\{p\ \underline{\text{is}}\ \mathsf{b}\}$. The least-fixed-point interpretation of this program contains two models. The first, $(p \mapsto \mathsf{just}\ \mathsf{b})$, obviously corresponds to the unique solution. The second, $(p \mapsto \mathsf{noneOf}\ \{\mathsf{b}\})$, does not correspond to any solution.

This is a relatively straightforward issue to resolve: we will only expect *positive* models (Definition 4.7) to correspond to solutions.

Another challenge for connecting the step semantics and the least-fixed-point interpretation has to do with infinite choice sets and constraint databases.

Example 5.11. The translated answer set program with no finite grounding that we showed in Figure 4 makes for an interesting example. All but one of the models in the least-fixed-point interpretation of this program are finite. Here is one finite model, corresponding to the visiting only 0 and then stopping:

$$(\mathit{visit}(\mathsf{z}) \mapsto \mathsf{just}\ \mathsf{tt},\ \ \mathit{more}(\mathsf{z}) \mapsto \mathsf{just}\ \mathsf{ff},\ \ \mathit{stop}(\mathsf{z}) \mapsto \mathsf{just}\ \mathsf{tt})$$

That constraint database corresponds to the fact set $\{\mathit{visit}(\mathsf{z})\ \underline{\text{is}}\ \mathsf{tt}, \mathit{more}(\mathsf{z})\ \underline{\text{is}}\ \mathsf{ff}, \mathit{stop}(\mathsf{z})\ \underline{\text{is}}\ \mathsf{tt}\}$, which is a solution according to Definition 2.11.

In addition to countably infinite constraint databases that correspond to solutions, the least-fixed-point interpretation of the program in Figure 4 also includes a single infinite constraint database that does not correspond to any solution:

$$\bigvee_{i \in \mathbb{N}} (\mathit{visit}(\mathsf{s}^i(\mathsf{z})) \mapsto \mathsf{just}\ \mathsf{tt},\ \ \mathit{stop}(\mathsf{s}^i(\mathsf{z})) \mapsto \mathsf{just}\ \mathsf{ff},\ \ \mathit{more}(\mathsf{s}^i(\mathsf{z})) \mapsto \mathsf{just}\ \mathsf{tt})$$

Example 5.11 demonstrates an advantage the least-fixed-point interpretation: it lets us reason mathematically about the meaning of programs that cannot be reached in finitely many steps. The least-fixed-point interpretation of datalog confers analogous advantages: a simple forward-chaining interpreter cannot fully evaluate the datalog program with two rules $(p(\mathsf{z}) \longleftarrow)$ and $(p(\mathsf{s}(x)) \longleftarrow p(x))$, but the least-fixed-point interpretation assigns the program a canonical interpretation as the infinite set $\{p(\mathsf{s}^n(\mathsf{z})) \mid n \in \mathbb{N}\}$.

Because we defined solutions to be finite sets, we have to account for the fact that the least fixed point interpretation of a finite-choice logic program may contain constraint databases that are not finite, and therefore do not correspond to solutions.

The following theorem establishes that there are no other caveats beyond the two highlighted by Examples 5.10 and 5.11:

Theorem 5.12. *For $D \in DB$, the following are equivalent:*

(1) *D is a solution to P by Definition 2.11.*
(2) *$D \in \text{lfp}\,\mathcal{T}_P$ and D is positive and finite.*

$$\begin{array}{ll}
\{\ (p_1 \mapsto \mathsf{just}\ \mathsf{b})\ \} \lor & \{\ (p_1 \mapsto \mathsf{just}\ \mathsf{b},\ p_2 \mapsto \mathsf{just}\ \mathsf{b},\ p_3 \mapsto \mathsf{just}\ \mathsf{b}),\\
\{\ (p_2 \mapsto \mathsf{just}\ \mathsf{b}),\ (p_2 \mapsto \mathsf{just}\ \mathsf{c})\ \} \lor & \quad (p_1 \mapsto \mathsf{just}\ \mathsf{b},\ p_2 \mapsto \mathsf{just}\ \mathsf{b},\ p_3 \mapsto \mathsf{noneOf}\ \{\mathsf{b}\}),\\
\{\ (p_3 \mapsto \mathsf{just}\ \mathsf{b}),\ (p_3 \mapsto \mathsf{noneOf}\ \{\mathsf{b}\})\ \} & \quad (p_1 \mapsto \mathsf{just}\ \mathsf{b},\ p_2 \mapsto \mathsf{just}\ \mathsf{c},\ p_3 \mapsto \mathsf{just}\ \mathsf{b}),\\
 & \quad (p_1 \mapsto \mathsf{just}\ \mathsf{b},\ p_2 \mapsto \mathsf{just}\ \mathsf{c},\ p_3 \mapsto \mathsf{noneOf}\ \{\mathsf{b}\})\ \}
\end{array}$$

Fig. 6. Two views of the same choice set. On the right, the choice set is represented straightforwardly as a set of pairwise-incompatible constraint databases. On the left, the choice set is represented as the least upper bound of singular choice sets.

Proof. Details are in the supplemental materials (Appendix C), but we will provide an outline here:

Going from (1) to (2) we first establish that any step which is precluded by the immediate consequence operator (as discussed in the introduction to Section 4) is actually a dead end, which means that productive steps taken towards a solution are always suitably "within" the set of immediate consequences. We then observe that all solutions are models, fixed points as defined by Definition 5.2, and are therefore bounded below by some minimal model in lfp $\mathcal{T}_P$. But any step sequence deriving a solution must be bounded above by a constraint database in lfp $\mathcal{T}_P$, and this squeezes solutions into membership in lfp $\mathcal{T}_P$.

Going from (2) to (1) we establish that any finite constraint database in lfp $\mathcal{T}_P$ belongs to a finite iteration of $\mathcal{T}_P$, and the action of each of those finitely-many iterations can be simulated with finite step sequences. □

6 Abstract Algorithm

Choice sets are critical both in defining the immediate consequence operators τ_P and $\mathcal{T}_P$ and in defining the meaning of a finite-choice logic program as lfp $\mathcal{T}_P$. However, the infrastructure of choice sets is more heavyweight than was strictly required to define τ_P. For any program P and constraint database D, we can express $\tau_P(D)$ in a simplified form, as the least upper bound of *singular* choice sets:

Definition 6.1. A choice set C is *singular* to an attribute a when, for all $D \in C$, we have that $D[a'] = \mathsf{noneOf}\ \varnothing$ whenever $a' \neq a$.

The least upper bound of multiple choice sets singular to a is also a choice set singular to a, so we can view the output of immediate consequence as a *map from attributes to singular choice sets.* This critical shift in perspective is illustrated in Figure 6 and is specified formally in supplemental materials (Appendix C).

Note that while $\tau_P(D)$ can always be written as the least upper bound of singular choice sets, not all choice sets can be described this way. A simple example is the program from Figure 1, whose interpretation is given by this choice set:

$$\{\ (p \mapsto \mathsf{just}\ \mathsf{ff},\ q \mapsto \mathsf{just}\ \mathsf{tt}),\ (p \mapsto \mathsf{just}\ \mathsf{tt},\ q \mapsto \mathsf{just}\ \mathsf{ff})\ \}$$

6.1 An Algorithm for Exploring the Interpretation of a Finite-Choice Logic Program

This nondeterministic algorithm attempts to return a single element from lfp $\mathcal{T}_P$.

- Initially, let D be $\bot_{DB}$.
- While $\tau_P(D) \neq \{D\}$ and $\tau_P(D) \neq \varnothing$:

(1) Interpret $\tau_P(D)$ as a map from attributes a to choice sets singular in a.

(2) Nondeterministically pick some attribute a where $\tau_P(D)[a] \neq \{(a \mapsto D[a])\}$.
(The loop guard $\tau_P(D) \neq \{D\}$ ensures some such a exists.)
(3) Nondeterministically pick some $(a \mapsto c)$ in $\tau_P(D)[a]$.
(The loop guard $\tau_P(D) \neq \emptyset$ ensures that $\tau_P(D)[a] \neq \emptyset$.)
(4) Modify D by setting $D[a]$ to be c.

- Successfully return D if $\tau_P(D) = \{D\}$, and return failure if $\tau_P(D) = \emptyset$.

This algorithm captures the intuition discussed at the start of Section 4: we can use the immediate consequences of a database to guide the process of picking the next step in the step-by-step computation of solutions to a finite-choice logic program.

Theorem 6.2. *For positive constraint databases D, the following are equivalent:*

(1) *The algorithm described above may successfully return D.*
(2) *$D \in \operatorname{lfp} \mathcal{T}_P$ and D is finite.*

This a corollary of the lemmas used to establish Theorem 5.12, which are presented in the supplemental materials (Appendix C).

The key to efficient evaluation of this algorithm is that we can incrementally update and efficiently query the portion of $\tau_P(D)$ where $\tau_P(D)[a] \neq \{(a \mapsto D[a])\}$. This is not original to finite-choice logic programming or our implementation: maintaining a data structure capturing all immediate consequences is perhaps *the* fundamental move for efficient implementation of semi-naive, tuple-at-a-time forward-chaining evaluation of logic programs. To give two examples, this is precisely the purpose of the queue Q in [McAllester 2002] and of the *agenda* A in [Eisner 2023]. Our setting is novel because the immediate consequence of a partial solution is not another deterministically-defined partial solution — as it is for McAllester, Eisner, and all other work we are aware of — but a pairwise-incompatible set of partial solutions, which we explore nondeterministically.

6.2 Resolving Nondeterminism

There are two nondeterministic choice points in the abstract algorithm described in Section 6.1: the choice of an attribute a in step 2 of the loop, and the choice from $\tau_P(D)[a]$ in step 3 of the loop. The nondeterministic choice in step 2 is common to all similar semi-naive, tuple-at-a-time forward-chaining evaluation algorithms, but the nondeterministic choice in step 3 is not.

We conjecture that the selection of an attribute (in step 2 of the abstract algorithm's loop) isn't "lossy:" when the algorithm might pick one of two attributes, the choice of one won't preclude finding any answer that the program might have returned if it had instead picked the other attribute. This means that a backtracking version of our algorithm that searches for all solutions doesn't need to reconsider the choices made in step 2.

While each *individual* attribute choice cannot cut off the path to any particular solution, a *systematic* bias in the way attributes are selected can interfere with the algorithm's nondeterministic completeness. Nevertheless, we choose for our implementation to be systematically biased by the *deduce-then-choose* strategy introduced in Section 3.2. In the context of our abstract algorithm, the deduce-then-choose strategy always picks an attribute a where $\tau_P(D)[a]$ is a singleton when such a choice is possible. This strategy does force non-termination on the abstract algorithm where it might otherwise be able to terminate, as the following example shows:

Example 6.3. Consider the following five-rule finite-choice logic program:

$$p \ \underline{\text{IS}} \ \{\mathsf{red}\} \leftarrow \tag{37}$$

$$q \ \underline{\text{IS}} \ \{\mathsf{blue}, \mathsf{yellow}\} \leftarrow num(\mathsf{s}(\mathsf{s}(\mathsf{z}))) \tag{38}$$

$$p \ \underline{\text{IS}} \ \{x\} \leftarrow q \ \underline{\text{IS}} \ x \tag{39}$$

$$num(\mathsf{z}) \leftarrow \tag{40}$$

$$num(\mathsf{s}(x)) \leftarrow num(x) \tag{41}$$

This program has no solutions — its least-fixed-point interpretation is the empty set — and it is possible for the abstract algorithm in Section 6.1 to finitely return failure or to loop endlessly, depending on how nondeterministic choices are resolved. However, the additional constraint imposed by the deduce-then-choose strategy means that our algorithm cannot finitely return failure, and can only fail to terminate.

Unlike the choice of which attribute to consider in step 2 of the abstract algorithm, the choice in step 3 to pick a specific value necessarily cuts off possible solutions. However, we can turn our abstract algorithm into a partially correct algorithm for enumerating the elements of a least-fixed-point interpretation by backtracking over the choices made in step 3.

Alongside backtracking, we add in one final bias. While our abstract algorithm enumerates arbitrary members of lfp $\mathcal{T}_P$, we are interested only in *solutions*, which by Theorem 5.12 are the *positive* constraint databases in lfp $\mathcal{T}_P$. Accordingly, we restrict our backtracking strategy to prefer constraints of the form $(\mathsf{just}\ t)$ prior to backtracking. This means that if the algorithm returns without backtracking, the result will be a positive model.

This combination of strategies—the deduce-then-choose strategy, no backtracking over choices of attribute, and some form of backtracking over the choice of constraint while preferring positive constraints—means that we can think of an execution of the abstract algorithm as a process of expanding a (potentially infinite) tree of decisions.

Example 6.4. Consider the following five-rule finite-choice logic program:

$$p \ \underline{\text{IS}} \ \{\mathsf{tt}, \mathsf{ff}\} \leftarrow \tag{42}$$

$$q \ \underline{\text{IS}} \ \{\mathsf{tt}, \mathsf{ff}\} \leftarrow \tag{43}$$

$$r \ \underline{\text{IS?}} \ \mathsf{a} \leftarrow \tag{44}$$

$$r \ \underline{\text{IS}} \ \{\mathsf{b}, \mathsf{c}\} \leftarrow p \ \underline{\text{IS}} \ \mathsf{ff} \tag{45}$$

$$r \ \underline{\text{IS}} \ \{x\} \leftarrow p \ \underline{\text{IS}} \ x, q \ \underline{\text{IS}} \ x \tag{46}$$

Two of the many possible induced execution trees are show in Figure 7. The upper tree Figure 7 represents a execution where p is considered first in step 2 of the abstract algorithm's loop, and the lower tree represents an execution where r is considered first.

In the first example where p is considered first, when the algorithm decides to give p the value $\mathsf{just}\ \mathsf{tt}$, the next attribute considered is q, but when p is given the value $\mathsf{just}\ \mathsf{ff}$, the next attribute considered is r. This highlights that we are not insisting on a single global ordering of attributes.

6.3 Completeness of the Constrained Algorithm

Example 6.3 demonstrated that the restrictions we put on the nondeterministic algorithm in this section force the non-backtracking version of our algorithm into non-termination where it might have otherwise terminated signaling failure. It is an open question whether the restrictions in this section interfere with the nondeterministic completeness of the algorithm: in other words, we don't know whether there is a program P and a database $D \in$ lfp $\mathcal{T}_P$ such that the more restricted

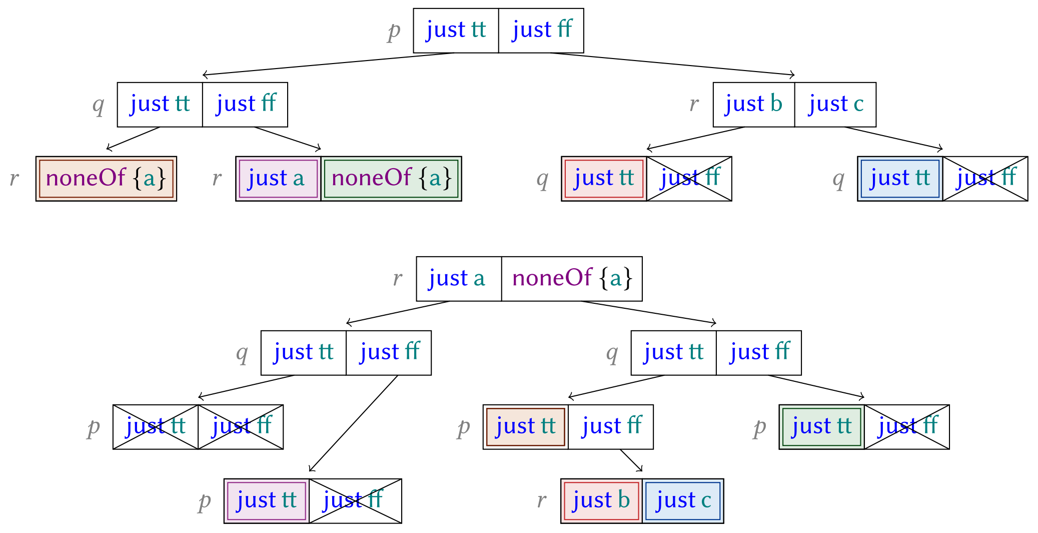


Fig. 7. Two trees of nondeterministic execution induced by different ways of selecting attributes in course of executing the finite-choice logic program from Example 6.4. Leaves representing failure-return are crossed out, and leaves representing successful terms are highlighted.

The algorithm potentially returns the same five models regardless of the order in which attributes are selected. The related models are outlined in the same color in the two trees. For example, the leaf that corresponds to returning the model ($p \mapsto$ just tt, $q \mapsto$ just tt, $r \mapsto$ noneOf {a}) is highlighted in brown and can be found on the left side of the upper tree and near the center of the lower tree.

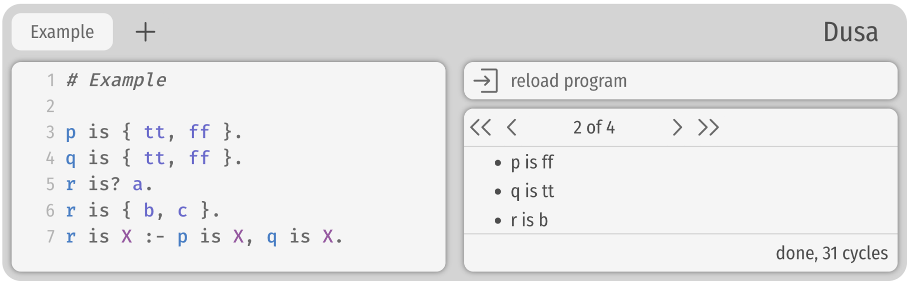
algorithm is unable return D. We conjecture that, even with the refinements we have discussed, any finite $D \in \text{lfp}\,\mathcal{T}_P$ can be reached with a suitable choice of attributes in step 2 of the algorithm's loop. Even if these restrictions do sacrifice completeness, we believe they are worthwhile. The deduce-then-choose strategy makes the performance of finite-choice logic programs vastly more predictable, as demonstrated by the examples in Section 3.

7 Implementation

Our implementation of the constrained abstract algorithm for finite-choice logic programming is called Dusa. The implementation supports three modes of interaction: a TypeScript API, a command-line program, and the browser-based editor pictured in Figure 8 and accessible at https://dusa.rocks/. All modes allow the client to request incremental enumeration of all solutions to a finite-choice logic program, and the first two modes allow the client to sample individual solutions [Simmons 2024].

Despite implementing the algorithm in Section 6.1, which enumerates finite models in lfp $\mathcal{T}_P$, our implementation discards all non-positive models and only exposes solutions—the positive models in the program's interpretation—to the client. That's why, while the interpretation of the program in Example 6.4 includes five different models, the web interface in Figure 8 only indicates the presence of four solutions.

Our implementation of finite-choice logic programming was initially designed with procedural generation and possibility-space exploration in mind. This makes our design considerations very similar to those that led Horswill to design CatSAT as a way to facilitate procedural generation in



```
# Example

p is { tt, ff }.
q is { tt, ff }.
r is? a.
r is { b, c }.
r is X :- p is X, q is X.
```

- p is ff
- q is tt
- r is b

Fig. 8. Online editor for the Dusa implementation of finite-choice logic programming at https://dusa.rocks/.

Unity games [Horswill 2018]. Horswill identifies four barriers to the use of answer set programming in video games, all of which are relevant to finite-choice logic programming and the Dusa implementation:

Designer Transparency. By this, Horswill refers to tools that allow designers to understand and manipulate programs. CatSAT does not directly address this barrier, and while Dusa has not been systematically tested with designers, our web interface does make it possible for non-experts to inspect, manipulate, and share programs without downloading or installing software (meeting many of the criteria for *casual creators* [Compton and Mateas 2015]).

Performance. CatSAT uses a simple implementation of the WalkSAT algorithm to get reasonable time performance and excellent memory performance on satisfability problems that are not excessively constrained. Dusa approaches performance from a different angle: our implementation is relatively memory-intensive, but we aim to provide *predictable* performance, which we discuss more in Section 7.1.

Run-Time Integration. Whereas answer set programming tools are primarily written as standalone programs, CatSAT is implemented as a DSL in C# to naturally integrate with programs in the Unity ecosystem. Dusa is implemented in TypeScript to facilitate integration with the web ecosystem.

Determinism. Horswill observes that most SAT solvers and answer set programming languages make it difficult or impossible to access suitably random behavior. Like CatSAT, the Dusa implementation defaults to randomness. After constraining the abstract algorithm as described in Section 6.2, our implementation resolves all remaining nondeterministic choices uniformly at random. Furthermore, rather than using a stack-based backtracking algorithm, our implementation creates in-memory representations of the trees described in Figure 7. After a solution is discovered, the algorithm prunes fully-explored parts of the tree and then returns to the root of the tree to begin random exploration again. The goal of this strategy is to avoid returning a second solution that is very similar to the first, a behavior that is commonly observed when exploring a state space with a stack-based (or depth-first) backtracking search.

While our strategy for randomized exploration does not truly sample the interpretation at random, in practice it has proven suitable for procedural generation. In our experience, Dusa also does a good job of locating "small" solutions for programs with an infinite interpretation like the one in Figure 4. However, when asked to enumerate multiple solutions for this specific program, Dusa will start by presenting smaller solutions and will then only present successively larger solutions, never returning to smaller solutions that were skipped along the way. This behavior, which Dusa shares with the Alpha implementation of ASP with lazy grounding [Weinzierl 2017], is unsuitable for procedural generation, and we hope to address it in future work.

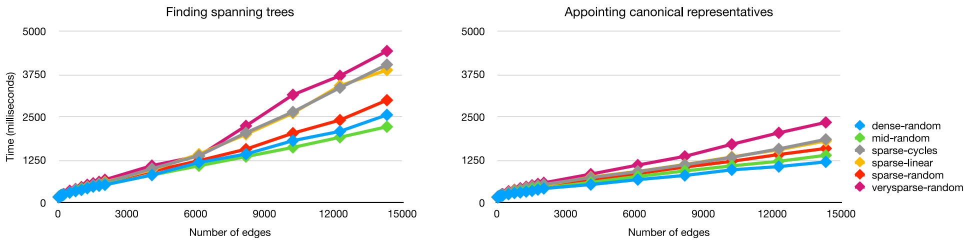


Fig. 9. Performance of the spanning tree program from Section 3.1 and the canonical representative algorithm from Section 3.2 on graphs with different characteristics. All data points are the median of three runs requesting a single solution.

7.1 Predictable Performance With Prefix Firings

The implementation of the abstract evaluation algorithm in Dusa is closely modeled after the semi-naive, tuple-at-a-time forward-chaining interpreter for datalog presented by McAllester [2002]. McAllester's algorithm was designed to facilitate a *cost semantics*, a way of reasoning about the run-time cost of evaluation without reasoning about the details of how the implementation works.

McAllester's cost semantics is based on counting the number of *prefix firings* in a solution. If a rule $H \leftarrow F$ has the premises $F = p_1(\overline{t_1})\ \underline{\text{IS}}\ v_1, \ldots, p_n(\overline{t_n})\ \underline{\text{IS}}\ v_n$, then a prefix firing for the database D is a unique variable-free instantiation of the first $i \leq n$ premises $p_1(\sigma\overline{t_1})\ \underline{\text{IS}}\ v_1, \ldots, p_i(\sigma\overline{t_i})\ \underline{\text{IS}}\ v_i$ that is satisfied in D. The central result of McAllester's work is that, assuming constant-time hashtable operations, a datalog program that returns the solution D_{final} can be evaluated in time proportional to the prefix firings for D_{final}. This cost semantics allows programmers to make decisions about how to write declarative programs: the McAllester cost semantics suggests defining the recursive rule for a transitive closure as $path(x, z) \leftarrow edge(x, y), path(y, z)$ rather than $path(x, z) \leftarrow path(x, y), path(y, z)$, because in a graph where the number of edges is proportional to n, the number of vertices, the former rule has $O(n^2)$ prefix firings and the latter has $O(n^3)$.

It would be nice to say that McAllester's cost semantics apply to Dusa programs that reach a solution without backtracking, but that isn't quite the case. McAllester's algorithm represents an extreme point in possible time-space tradeoffs: the result of every deduction is memoized in a hash table. (If one makes the standard assumption that hash table operations take constant time, this results in a space complexity equal to the time complexity.) In order to support our general backtracking strategy, our implementation uses AVL trees to implement the maps that McAllester's algorithm implements as hash tables. To apply a prefix-firing cost semantics to finite-choice logic programming, we will need to take one of two approaches: either we must adapt the cost semantics to account for the logarithmic factors involved in accessing functional maps, or we must describe a modified implementation of our algorithm that uses hash tables.

Nevertheless, the prefix-firing cost semantics is successful at predicting the behavior of Dusa programs that do not backtrack. The programs for spanning tree generation in Figure 2 and for canonical representative appointment in Figure 3 are expected, in any deduce-then-choose execution, to find some solution without backtracking, and solutions for both programs will have prefix firings proportional to the number of edges. Figure 9 demonstrates that the actual behavior of Dusa generally conforms to the running time predicted by the cost semantics, possibly with some additional logarithmic factors thrown in to account for the use of functional data structures.

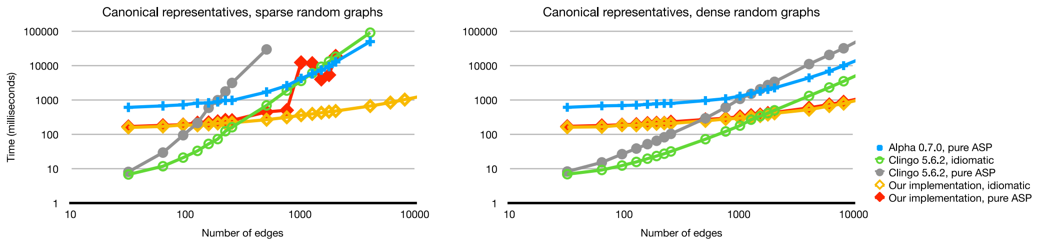


Fig. 10. Performance of Alpha, Clingo, and Dusa at finding canonical representatives. All data points are the median of three runs requesting a single solution, discarding runs exceeding 100 seconds.

7.2 Comparing Dusa to Answer Set Programming

There's not an obvious way to ask an answer set programming engine to run the spanning tree program from Section 3.1 or the canonical representative program from Section 3.2, because we have not defined a translation from finite-choice logic programs to answer set programs. However, solutions to the rooted spanning tree problem and the canonical representative problem can be expressed in answer set programming, and these answer set programs can be translated into finite-choice logic programs as discussed in Sections 2.3 and 3.3.

In Figure 10 we compare the canonical representative problem from Figure 3 against an idiomatic Clingo program that uses features like cardinaity constraints, as well as comparing the performance of a single "pure" answer set program executed in Clingo, in the Alpha implementation of ASP with lazy grounding [Weinzierl 2017], and in Dusa by direct translation of ASP to finite-choice logic programming. (The yellow lines in Figure 10 repeat the red and blue lines from the right-hand side of Figure 9.) In contrast with the predictable behavior shown in Figure 9, the performance of the other four solutions varies significantly depending on characteristics of the graph. In the head-to-head comparison, Dusa outperforms the state of the art on dense graphs but does worse as graphs get sparser.

The poor performance of Clingo in the head-to-head comparison can be attributed to the "grounding bottleneck" encountered by traditional approaches to ASP. Both Alpha's lazy grounding and Dusa's deduce-then-choose execution model represent ways of avoiding this grounding bottleneck, and this translates to better asymptotic performance for this example.

Compared with modern ASP implementations, the backtracking strategy used by the current Dusa implementation is incredibly naive, comparable to 1990s-era DPLL SAT solvers that lacked non-chronological backtracking. This lack of sophistication is almost certainly the cause of Dusa's relatively poor performance in head-to-head comparisons on sparse graphs. More generally, our implementation struggles on programs like the Boolean satisfiability example in Figure 5 that define a possibility space generically and then use constraints to whittle it down to a desired state. This *design-space sculpting* approach to modeling [Smith and Mateas 2011] is a natural way to express many problems: to implement procedural generation we can let every point in a grid be land or sea and demand that there be a path between two specific points, to implement graph coloring we can assign every node in a graph to one of five colors and demand that no two edge-connected nodes have the same color, and to implement the N-queens problem we can let every space on a N-by-N chessboard contain a queen or not and demand that there are N queens on the board and that there are no immediate avenues of attack. Variants of all these problems are considered in our extended benchmarking results, available in supplemental materials (Appendix D). In future work, we intend to adapt Dusa to better support sculptural finite-choice logic programming by adapting techniques such as conflict-driven nogood learning that have proven successful in similar settings [Gebser et al. 2012].

8 Related Work

Our least-fixed-point interpretation of finite-choice logic programming describes a domain-like structure for nondeterminism that draws inspiration from domain theory writ large [Scott 1982], particularly powerdomains for nondeterministic lambda calculi and imperative programs [Kennaway and Hoare 1980; Plotkin 1976; Smyth 1976]. Our partial order on choice sets matches the one used for Smyth powerdomains in particular [Smyth 1976]. Our construction requires certain completeness criteria on posets that differ from these and other domain definitions we have found in the literature.

Both the least-fixed-point interpretation and the step-sequence interpretation of finite-choice logic programming represent ways of reasoning about the incremental development of solutions. Because we provide a translation of answer set programming into finite-choice logic programming, these ways of reasoning about incremental development of solutions are applicable to ASP as well. Previous work in this area includes the *stable backtracking fixedpoint* algorithm described by Sacca and Zaniolo [1990]. (We interpret their algorithm as giving the first direct account for answer set programming without grounding, though it seems to us that this significant fact was not noticed or exploited by the authors or anyone else.) There has also been a disjunctive extension to datalog [Eiter et al. 1997] and characterizations of its stable models [Przymusinski 1991]; Leone et al. [1997] present an algorithm for incrementally deriving stable models for disjunctive datalog. Their approach is based on initially creating a single canonical model by letting some propositions be "partially true."

At the core of finite-choice logic programming is the propositional form $p(\bar{t})$ is v, which establishes a *functional dependency* from an attribute to a value. This mirrors the proposition form $R[x_1, \ldots, x_{n-1}] = x_n$ used to enforce integrity constraints in LogiQL [Aref et al. 2015]. Soufflé's functional dependencies allow programs to express multiple mutually contradictory conclusions [Hu et al. 2021], but as as discussed in Section 3.1, functional dependencies in the tradition of Krishnamurthy and Naqvi [1988] cannot express that the violation of a functional dependency should cause a potential solution to be rejected. Roughly speaking, LogiQL works like a finite-choice logic programming language that only has closed rules without multiple choices, and Soufflé works like a finite-choice logic programming language that has only open rules. These systems are less expressive than ours in this respect, but they can completely avoid expensive backtracking.

The partially-ordered set *Constraint* introduced in Section 4.2 suggests a relationship between finite-choice logic programming and other languages that assign non-Boolean values to propositions: values drawn from a semiring in weighted logic programming [Eisner et al. 2005], various interpretations of degree-of-truth in annotated logic programming [Kifer and Subrahmanian 1992] and bilattice-based logic programming [Fitting 1991], and probabilities in probabilistic logic programming [Sterling 1995]. As discussed in Section 4.3, we're unaware of work along these lines that can give an account for stable models without falling back on a variant of Gelfond and Lifschitz's syntactic transformation. However, the related mathematical structures shared across these approaches points towards generalizing *Constraint* to accept alternate partial orders, similar to the lattice-based functional dependencies seen in datalog extensions like Bloom [Conway et al. 2012], Flix [Madsen and Lhoták 2020], and Egglog [Zhang et al. 2023].

Fandinno et al. [2023] extend a translation due to Niemlä [2008] to formalize the "idea from folklore" that ASP can be based on an ASP choice rules along with integrity constraints. This is similar to our observation that choice, rather than negation, is a suitable foundation for answer set programming, though a limited (stratified) negation is still necessary in their translation. They present a conceptual operational semantics based on making *all* possible choices up-front, followed by a phase of deduction; this operational interpretation is intended as a pedagogical tool rather than the basis of a practical implementation.

9 Conclusion

We have introduced the theory and implementation of finite-choice logic programming, an approach to logic programming that has connections to answer set programming but that treats *choice*, not *negation*, as the fundamental primitive of nondeterminism. We establish that choice is a suitably expressive foundation by showing that answer set programming can be defined in terms of finite-choice logic programming.

We give a first definition of finite-choice logic programming in terms of nondeterministically augmenting a set of facts, a second definition as the unique least fixed point in a novel domain of mutually-exclusive models, and a proof that these two definitions agree on solutions.

Our Dusa implementation can enumerate solutions to finite-choice logic programs, and the runtime behavior of our implementation can often reliably (if approximately) be predicted by McAllester's cost semantics based on prefix firings. Because our implementation is not subject to the "grounding bottleneck" that affects mainstream answer set programming solvers, our implementation outperforms the state-of-the-art Clingo ASP implementation in many cases despite its extremely naive backtracking strategy.

In future work, we hope to firmly establish the status of nondeterministic completeness for the abstract algorithm as restricted by the deduce-then-choose strategy (as discussed in Section 6.3), investigate fair enumeration of solutions (as discussed in Section 7), and apply proven techniques from ASP solvers and Boolean satisfiability solvers to improve Dusa's performance on programs where the current implementation's simplistic backtracking strategy does not perform well (as discussed in Section 7.2). In addition, we plan to investigate the semantics of fragments of our language, both by giving a precise account for connecting open rules and languages with choice constructs like Soufflé's and by seeing if closed rules can account for disjunctive datalog. We also hope to generalize finite-choice logic programming as presented here, particularly by extending the partial order on *Constraint* to richer partial orders, as has been done in related systems, which we believe could expand the expressiveness of finite-choice logic programming to account for monotonic data aggregation [Ross and Sagiv 1992]. Finally, we hope to explore proof-theoretic accounts of finite-choice logic programming, which could support compositional explanation and provenance analysis for possibility spaces.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1846122. The genesis of this work was performed at the Recurse Center, and we are grateful for the Recurse Center for nurturing a space where people can think about ideas in programming.

We want to thank Richard Comploi-Taupe, who helped us understand the state of research on answer set programming with lazy grounding, and Ben Simner, who helped acquire reference materials on disjunctive logic programming. Many folks gave helpful feedback that improved this paper, particularly Mitch Wand, Emma Tosch, Jason Reed, Philip Zucker, David Renshaw, and six anonymous reviewers.

References

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in time and space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer, Oxford, UK, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices* 46, 4 (2011), 53–64. https://doi.org/10.1145/1988042.1988046

Keith L. Clark. 1978. Negation as Failure. In *Logic and Data Bases*, Hervé Gallaire and Jack Minker (Eds.). Springer US, Boston, MA, 293–322. https://doi.org/10.1007/978-1-4684-3384-5_11

Richard Comploi-Taupe, Gerhard Friedrich, Konstantin Schekotihin, and Antonius Weinzierl. 2023. Domain-Specific Heuristics in Answer Set Programming: A Declarative Non-Monotonic Approach. *J. Artif. Int. Res.* 76 (may 2023), 56 pages. https://doi.org/10.1613/jair.1.14091

Kate Compton and Michael Mateas. 2015. Casual Creators. In *Proceedings of the Sixth International Conference on Computational Creativity (ICCC 2015)*, Hannu Toivonen, Simon Colton, Michael Cook, and Dan Ventura (Eds.). Brigham Young University, Park City, Utah, 228–235. http://computationalcreativity.net/iccc2015/proceedings/10_2Compton.pdf

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged Configuration Using Feature Models. In *Software Product Lines*, Robert L. Nord (Ed.). Springer, Berlin, Heidelberg, 266–283. https://doi.org/10.1007/978-3-540-28630-1_17

Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative Programming. In *European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 15–29. https://doi.org/10.1007/3-540-36208-8_2

Chinmaya Dabral and Chris Martens. 2020. Generating explorable narrative spaces with answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 16. AAAI Press, Washington, USA, 45–51. https://doi.org/10.1609/aiide.v16i1.7406

Chinmaya Dabral, Emma Tosch, and Chris Martens. 2023. Exploring Consequences of Privacy Policies with Narrative Generation via Answer Set Programming. (2023). arXiv:2212.06719 Presented at Workshop on Programming Languages and the Law (ProLaLa@POPL).

Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. 2009. GASP: Answer Set Programming with Lazy Grounding. *Fundam. Inf.* 96, 3 (2009), 297–322.

Joris Dormans and Sander Bakkes. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 216–228. https://doi.org/10.1109/TCIAIG.2011.2149523

Jason Eisner. 2023. Time-and-Space-Efficient Weighted Deduction. *Transactions of the Association for Computational Linguistics* 11 (08 2023), 960–973. https://doi.org/10.1162/tacl_a_00588

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling Comp Ling: Weighted Dynamic Programming and the Dyna Language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, Raymond Mooney, Chris Brew, Lee-Feng Chien, and Katrin Kirchhoff (Eds.). Association for Computational Linguistics, Vancouver, British Columbia, Canada, 281–290. https://aclanthology.org/H05-1036

Thomas Eiter, Georg Gottlob, and Heikki Mannila. 1997. Disjunctive datalog. *ACM Transactions on Database Systems (TODS)* 22, 3 (1997), 364–418. https://doi.org/10.1145/261124.261126

Jorge Fandinno, Seemran Mishra, Javier Romero, and Torsten Schaub. 2023. *Answer Set Programming Made Easy*. Springer, Cham, 133–150. https://doi.org/10.1007/978-3-031-31476-6_7

Melvin Fitting. 1991. Bilattices and the semantics of logic programming. *The Journal of Logic Programming* 11, 2 (1991), 91–116. https://doi.org/10.1016/0743-1066(91)90014-G

Melvin Fitting. 1993. The family of stable models. *The Journal of Logic Programming* 17, 2 (1993), 197–225. https://doi.org/10.1016/0743-1066(93)90031-B

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1 (2019), 27–82. https://doi.org/10.1017/S1471068418000054

Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011. Potassco: The Potsdam answer set solving collection. *AI Communications* 24, 2 (2011), 107–124. https://doi.org/10.3233/AIC-2011-0491

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187-188 (2012), 52–89. https://doi.org/10.1016/j.artint.2012.04.001

Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of International Logic Programming Conference and Symposium*, Robert Kowalski and Kenneth A. Bowen (Eds.). MIT Press, 1070–1080.

Fosca Giannotti, Dino Pedreschi, and Carlo Zaniolo. 2001. Semantics and Expressive Power of Nondeterministic Constructs in Deductive Databases. *J. Comput. System Sci.* 62, 1 (2001), 15–42. https://doi.org/10.1006/jcss.1999.1699

Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C Pierce. 2023. Reflecting on Random Generation. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 322–355. https://doi.org/10.1145/3607842

Harrison Goldstein and Benjamin C Pierce. 2022. Parsing randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 89–113. https://doi.org/10.1145/3563291

Sergio Greco and Carlo Zaniolo. 2001. Greedy algorithms in Datalog. *Theory and Practice of Logic Programming* 1, 4 (2001), 381–407. https://doi.org/10.1017/S1471068401001090

Ian Horswill. 2018. CatSAT: A Practical, Embedded, SAT Language for Runtime PCG. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14, 1 (Sep. 2018), 38–44. https://doi.org/10.1609/aiide.v14i1.13026

Xiaowen Hu, Joshua Karp, David Zhao, Abdul Zreika, Xi Wu, and Bernhard Scholz. 2021. The Choice Construct in the Soufflé Language. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 163–181. https://doi.org/10.1007/978-3-030-89051-3_10

Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. 2016. Grounding and Solving in Answer Set Programming. *AI Magazine* 37, 3 (Oct. 2016), 25–32. https://doi.org/10.1609/aimag.v37i3.2672

J.R. Kennaway and C.A.R. Hoare. 1980. A theory of nondeterminism. In *Automata, Languages and Programming*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer, Berlin, Heidelberg, 338–350. https://doi.org/10.1007/3-540-10003-2_82

Michael Kifer and V.S. Subrahmanian. 1992. Theory of generalized annotated logic programming and its applications. *The Journal of Logic Programming* 12, 4 (1992), 335–367. https://doi.org/10.1016/0743-1066(92)90007-P

Ekaterina Komendantskaya and Anthony Karel Seda. 2009. Sound and Complete SLD-Resolution for Bilattice-Based Annotated Logic Programs. *Electronic Notes in Theoretical Computer Science* 225 (2009), 141–159. https://doi.org/10.1016/j.entcs.2008.12.071

Ravi Krishnamurthy and Shamim Naqvi. 1988. Non-Deterministic Choice in Datalog. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, C. Beeri, J.W. Schmidt, and U. Dayal (Eds.). Morgan Kaufmann, 416–424. https://doi.org/10.1016/B978-1-4832-1313-2.50038-X

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2017. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/10.1145/3158133

Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011)*. IEEE Computer Society, Los Alamitos, CA, USA, 143–150. https://doi.org/10.1109/VLHCC.2011.6070391

Nicola Leone, Pasquale Rullo, and Francesco Scarcello. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and computation* 135, 2 (1997), 69–112. https://doi.org/10.1006/inco.1997.2630

Beidi Li, Jochen Teizer, and Carl Schultz. 2020. Non-monotonic Spatial Reasoning for Safety Analysis in Construction. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/3414080.3414096

How Khang Lim, Avishkar Mahajar, Martin Strecker, and Meng Weng Wong. 2022. Automating defeasible reasoning in law with answer set programming. *CEUR* 3193 (2022).

Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the masses: programming with first-class datalog constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428193

David McAllester. 2002. On the complexity analysis of static analyses. *J. ACM* 49, 4 (July 2002), 512–537. https://doi.org/10.1145/581771.581774

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic* 51, 1 (1991), 125–157. https://doi.org/10.1016/0168-0072(91)90068-W

Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. 2015. Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*. IEEE, 207–212. https://doi.org/10.1109/CEEC.2015.7332726

Ilkka Niemelä. 2008. Answer Set Programming without Unstratified Negation. In *Logic Programming*, Maria Garcia de la Banda and Enrico Pontelli (Eds.). Springer, Berlin, Heidelberg, 88–92. https://doi.org/10.1007/978-3-540-89982-2_15

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, NY, USA, 966–980. https://doi.org/10.1145/3519939.3523707

Gordon D. Plotkin. 1976. A powerdomain construction. *SIAM J. Comput.* 5, 3 (1976), 452–487. https://doi.org/10.1137/0205035

Teodor C Przymusinski. 1991. Stable semantics for disjunctive programs. *New generation computing* 9, 3-4 (1991), 401–424. https://doi.org/10.1007/BF03037171

Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic aggregation in deductive databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '92)*. Association for Computing Machinery, New York, NY, USA, 114–126. https://doi.org/10.1145/137097.137852

Domenico Sacca and Carlo Zaniolo. 1990. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville, USA) *(PODS '90)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/298514.298572

Dana S Scott. 1982. Domains for denotational semantics. In *Automata, Languages and Programming: Ninth Colloquium Aarhus, Denmark, July 12–16, 1982 9*. Springer, Berlin, Heidelberg, 577–610. https://doi.org/10.1007/BFb0012801

Eric L Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type targeted testing. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, Berlin, Heidelberg, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33

Noor Shaker, Julian Togelius, and Mark J Nelson. 2016. *Procedural content generation in games*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-42716-4

Tanya Short and Tarn Adams. 2017. *Procedural generation in game design*. CRC Press.

Robert Simmons. 2024. *Dusa implementation, examples, and benchmarking*. Zenodo. https://doi.org/10.5281/zenodo.13983457

Robert J. Simmons and Frank Pfenning. 2008. Linear Logical Algorithms. In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, Berlin, Heidelberg, 336–347. https://doi.org/10.1007/978-3-540-70583-3_28

Patrik Simons, Ilkka Niemelä, and Timo Soininen. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138, 1 (2002), 181–234. https://doi.org/10.1016/S0004-3702(02)00187-X Knowledge Representation and Logic Programming.

Anthony J Smith and Joanna J Bryson. 2014. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*.

Adam M Smith, Eric Butler, and Zoran Popovic. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Foundations of Digital Games*. 221–228.

Adam M Smith and Michael Mateas. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200. https://doi.org/10.1109/TCIAIG.2011.2158545

Michael B Smyth. 1976. Powerdomains. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, Berlin, Heidelberg, 537–543. https://doi.org/10.1007/3-540-07854-1_226

Leon S. Sterling. 1995. *A Statistical Learning Method for Logic Programs with Distribution Semantics*. MIT Press, 715–729. https://doi.org/10.7551/mitpress/4298.003.0069

Adam Summerville, Chris Martens, Ben Samuel, Joseph Osborn, Noah Wardrip-Fruin, and Michael Mateas. 2018. Gemini: Bidirectional generation and analysis of games via ASP. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14, 1 (2018), 123–129. https://doi.org/10.1609/aiide.v14i1.13013

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math* 5, 2 (1955), 285–309.

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM* 38, 3 (jul 1991), 619–649. https://doi.org/10.1145/116825.116838

Antonius Weinzierl. 2017. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *Logic Programming and Nonmonotonic Reasoning*, Marcello Balduccini and Tomi Janhunen (Eds.). Springer International Publishing, Cham, 191–204. https://doi.org/10.1007/978-3-319-61660-5_17

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492. https://doi.org/10.1145/3591239

Received 2024-07-09; accepted 2024-11-07