Generative Datalog with Stable Negation

Mario Alviano
University of Calabria
alviano@mat.unical.it

Michael Morak
University of Klagenfurt
michael.morak@aau.at

Matthias Lanzinger
University of Oxford
matthias.lanzinger@cs.ox.ac.uk

Andreas Pieris
University of Edinburgh & University of Cyprus
apieris@inf.ed.ac.uk

ABSTRACT
Extending programming languages with stochastic behaviour such as probabilistic choices or random sampling has a long tradition in computer science. A recent development in this direction is a declarative probabilistic programming language, proposed by Bárány et al. in 2017, which operates on standard relational databases. In particular, Bárány et al. proposed generative Datalog, a probabilistic extension of Datalog that allows sampling from discrete probability distributions. Intuitively, the output of a generative Datalog program on an input database is a probability space over the minimal models of Datalog, the so-called possible outcomes. This is a natural generalization of the (deterministic) semantics of Datalog, where the output of a program on a database is their unique minimal model. A natural question to ask is how generative Datalog can be enriched with the useful feature of negation, which in turn leads to a strictly more expressive declarative probabilistic programming language. In particular, the challenging question is how the probabilistic semantics of generative Datalog with negation can be robustly defined. Our goal is to provide an answer to this question by interpreting negation according to the stable model semantics.

CCS CONCEPTS
• Theory of computation → Constraint and logic programming: Database query languages (principles); Incomplete, inconsistent, and uncertain databases; • Mathematics of computing → Probabilistic representations.

KEYWORDS
probabilistic programming; generative Datalog; negation; stable model semantics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODS ’23, June 18–23, 2023, Seattle, WA, USA.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8757-3/23/06...
https://doi.org/10.1145/3584372.3588656

1 INTRODUCTION
Extending programming languages with stochastic behaviour, such as probabilistic choices or random sampling, has a long tradition in computer science that goes back in the 70s and 80s [19, 27]. During the last decade or so, there was a significant effort in developing dedicated probabilistic programming languages (for example, Pyro [4], Stan [6], Church [14], R2 [22], Figaro [23], and Anglican [29], to name a few) that allow the specification and “execution”, via probabilistic inference, of sophisticated probabilistic models.

A recent development in this direction is a purely declarative probabilistic programming language based on Datalog (PPDL), proposed by Bárány et al. [3], which operates on standard relational databases. A PPDL program consists of the generative and the constraint component. The generative component is essentially a rule-based program written in a probabilistic extension of Datalog, called generative Datalog, that allows sampling from discrete probability distributions. Intuitively, the output of a generative Datalog program on an input database is a probability space over the minimal models of Datalog, and the so-called possible outcomes. Typically, the probability space over the possible outcomes obtained via such a generative process is called prior distribution. The probabilistic semantics of generative Datalog can be seen as a natural generalization of the (deterministic) semantics of Datalog, where the output of a program on a database is their unique minimal model. Now, the constraint component allows one to pose logical constraints that the relevant possible outcomes should satisfy and, semantically, transforms the prior distribution into the subspace obtained by conditioning on the constraints.

Generative Datalog is indeed a powerful probabilistic formalism that allows for defining a rich family of probabilistic models, while retaining the key property of declarativity, that is, the independence of the order in which the rules are executed. On the other hand, as discussed below, there are simple probabilistic models that cannot be defined using generative Datalog due to its monotonic nature.

Example 1.1 (Network Resilience). We consider a network of routers and assume that some of them have been infected by a malware that interrupts the routing, and it also attempts to infect neighbouring routers with a success rate of 10%. We say that such a network is dominated by the malware if all routers are infected or isolated in the network (i.e., connected only to infected routers). In such a scenario, we would like to be able to predict how likely it is for the network to be dominated. The propagation of the malware can be encoded using the generative Datalog rule

Infected(x, true), Connected(x, y) → Infected(y, Flip(0.1))
meaning that if the router \( x \) is infected and \( x \) is connected with \( y \), then \( y \) is also infected with probability 0.1; that is, the probabilistic term \( \text{Flip}(0.1) \) becomes \text{true} with probability 0.1 or \text{false} with probability 0.9. However, the domination of the network is a non-monotonic property in the sense that expanding a dominated network may result in a non-dominated one, whereas generative Datalog is inherently monotonic. Hence, although the propagation of the malware can be encoded via generative Datalog, the likelihood of domination cannot be predicted via this formalism.

With the aim of overcoming the above weakness of generative Datalog, we enrich it with the useful feature of designation as failure, interpreted according to the standard stable model semantics stemming from logic programming [12]. Having such a formalism in place, we could predict the likelihood of a network being dominated using the probabilistic rule from Example 1.1 and the rules

\[
\text{Router}(x) \quad \neg\text{Infected}(x, \text{true}) \rightarrow \text{Uninfected}(x)
\]
\[
\text{Uninfected}(x), \text{Uninfected}(y), \text{Connected}(x,y) \rightarrow \text{Fail},
\]

which essentially state that a router which is not infected is uninfected, and whenever two uninfected routers are connected in the network, then the malware fails to dominate the network.

**Our Contributions.** Adding stable negation to generative Datalog comes with several technical complications, discussed in Section 4, that need to be studied and understood in order to define a robust semantics. This is precisely the goal of the present work. Our main contributions can be summarized as follows:

- In Section 4, we introduce a formal semantics for generative Datalog with stable negation. This cannot simply be done by following the development underlying the semantics of positive generative Datalog from [3] and considering stable models instead of minimal models. In fact, with negation, it is not enough to consider single models, we need to think of possible outcomes as sets of stable models. These sets are represented symbolically via ground programs that explicitly encode the underlying probabilistic choices. The latter is crucial since, due to non-monotonicity, the probabilistic choices are not necessarily reflected in the resulting set of stable models. Hence, a set of stable models may not carry enough information to allow us to calculate its probability.

- In Section 5, we show that the probabilistic semantics from Section 4 can be equivalently defined via a fixpoint procedure. We introduce a new chase procedure that operates on ground programs (rather than on databases, as in the standard chase procedure), which we then use to define a chase-based probability space that leads to the desired fixpoint semantics. Interestingly, the semantics is independent of the order in which the rules are executed.

- Finally, in Section 6, we concentrate on the central class of generative Datalog programs with stratified negation and provide further evidence of the robustness of our semantics.

## 2 RELATED WORK

The probabilistic programming and the probabilistic database communities have developed several different models and systems that allow to specify probability distributions over data. Since the paper by Bárány et al. [3], which introduced generative Datalog, already presents a thorough discussion on related work (see Section 7), we proceed to discuss only the related work that is closer to our work.

Conceptually closer to generative Datalog (with or without negation) are languages studied in statistical relational AI (StarAI) [24], which aim at the combination of predicate logic and probabilities. Here are some prominent examples of such languages:

- **ProbLog** [9, 25] is a probabilistic extension of Prolog where standard Prolog rules can be annotated with a probability value (i.e., it allows uncertainty at the level of rules). Hybrid ProbLog [17] is an extension of ProbLog that allows continuous attribute-level uncertainty in rule-heads.

- **Probabilistic Answer Set Programming** [7] extends the well-known logic programming language of Answer Set Programming [12, 13] with probabilistic facts. The semantics of this language is defined via upper and lower probability bounds for stable models. A number of similar extensions have been proposed in the literature based on the general idea of annotating facts or rules with probabilities [2, 20, 30]. A more complete overview is given by Cozman and Maulá [7].

- **Markov Logic Networks** (MLNs) [26] combine first-order logic and Markov Networks, and they describe joint distributions of variables based on weighted first-order constraints. Hybrid MLNs [31] extend MLNs to continuous distributions. Infinite MLNs [28] allow for countably infinitely many variables with countable domains.

- **Probabilistic Soft Logic** [18] is another statistical relational learning framework that combines first-order logic and probabilistic graphical models that specifies joint distributions with weighted rules, but also “soft” truth values.

- Another formal language that combines first-order logic and probabilistic graphical models, which also supports continuous distributions, is Bayesian Logic [21], which in turn builds on Bayesian Networks.

- A probabilistic extension of Datalog (a family of knowledge representation languages that enrich Datalog with features such as value invention and equality in head-rules [5]) that is based on MLNs as underlying probabilistic semantics has been proposed in [15]. Various probabilistic extensions of plain Datalog can be also found in [8, 10, 11].

Although the formalisms discussed above (as well as those discussed in Section 7 of [3]), might share individual features with generative Datalog (with or without negation), the combination of plain Datalog with probabilistic programming languages should be attributed to [3]. Moreover, the fact that generative Datalog (with or without negation) can express attribute-level uncertainty in rule-heads (via the probabilistic terms appearing in rule-heads) differs from other probabilistic extensions of Datalog, where the uncertainty is at the at level of rules specified by attaching probabilities to them (like, for example, the languages from [15] and [8, 10, 11]).

## 3 PRELIMINARIES

We recall the basics on relational databases, TGDs with stable negation, and probability spaces. We assume the disjoint countably infinite sets \( C \) and \( V \) of constants and variables, respectively; we refer to constants and variables as terms. For \( n > 0 \), let \( [n] = \{1, \ldots, n\} \).
Relational Databases. A (relational) schema $S$ is a finite set of relation names (or predicates) with associated arity: we write $ar(R)$ for the arity of a predicate $R$. A (relational) atom $\alpha$ over $S$ is an expression of the form $R(t_1, \ldots, t_n)$, where $R \in S$, $ar(R) = n$, and $t_i \in C \cup V$ for each $i \in [n]$. A literal over $S$ is either an atom over $S$ (i.e., a positive literal), or an atom preceded by the negation symbol $\neg$ (i.e., a negative literal). An instance of $S$ is a (possibly infinite) set of atoms over $S$ that mention only constants, while a database of $S$ is a finite instance of $S$. We write $dom(I)$ for the set of constants occurring in an instance $I$. For simplicity, in the rest of the paper, we assume that the constants of $C$ are translatable into real numbers. Therefore, whenever we refer to a constant of $C$, we actually refer to its translation into a real number.

TGDs with Stable Negation. A tuple-generating dependency with negation (or simply TGD\(^\neg\)) over a schema $S$ is an expression
\[
\forall \bar{x} \forall \bar{y} (\psi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \bar{\psi}(\bar{x}, \bar{z})),
\]
where $\psi(\bar{x}, \bar{y})$ (resp., $\bar{\psi}(\bar{x}, \bar{z})$) is a quantifier-free conjunction of literals (resp., atoms) over $S$ with variables from $\bar{x} \cup \bar{y}$ (resp., $\bar{x} \cup \bar{z}$), and each variable in a negative literal occurs also in a positive literal; the latter condition is known as safety. For brevity, we will write $\sigma$ as $\psi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \bar{\psi}(\bar{x}, \bar{z})$, and use comma instead of $\wedge$ for joining atoms.

For our purposes, although syntactically a TGD\(^\neg\) is a first-order sentence, semantically it is not since the negation should be interpreted as stable negation rather than classical negation. We proceed to recall the formal definition as given in [1]. The high-level idea is, given an (infinitary) TGD\(^\neg\) program $\Sigma$, to construct a second-order sentence $SM[\Sigma]$ that encodes the intention underlying the stable model semantics, and then define the stable models of $\Sigma$ as the classical models of $SM[\Sigma]$. Consider an (infinitary) TGD\(^\neg\) program $\Sigma$. Let $R = (R_1, \ldots, R_n)$ be the list of predicates of $sch(\Sigma)$ in some fixed order (e.g., lexicographic order), and $X_\Sigma = (X_\Sigma^1, \ldots, X_\Sigma^n)$ be a list of $n$ distinct predicate variables. For a literal $\ell$ occurring in $\Sigma$, let
\[
rep_{\Sigma}(\ell) = \begin{cases} X_\Sigma^i(i), & \text{if } \ell = R_i(i), \\ \neg R_i(i), & \text{if } \ell = \neg R_i(i). \end{cases}
\]
We define $rep_{\Sigma}(\Sigma)$ as the (infinitary) TGD\(^\neg\) program obtained by applying $rep_{\Sigma}(\cdot)$ to every literal in $\Sigma$. The sentence $SM[\Sigma]$ is
\[
\bigwedge_{\sigma \in SM[\Sigma]} (\sigma \wedge \neg \exists X_{\Sigma} ((X_{\Sigma} < R) \wedge rep_{\Sigma}(\Sigma))),
\]
where $X_{\Sigma} < R$ is the sentence
\[
\bigwedge_{i=1}^{n} \forall x_1 \ldots \forall x_{ar(R_i)} \left( X_{\Sigma}^i(x_1, \ldots, x_{ar(R_i)}) \rightarrow R_i(x_1, \ldots, x_{ar(R_i)}) \right) \wedge
\left( \bigwedge_{i=1}^{n} \forall x_1 \ldots \forall x_{ar(R_i)} \left( R_i(x_1, \ldots, x_{ar(R_i)}) \rightarrow X_{\Sigma}^i(x_1, \ldots, x_{ar(R_i)}) \right) \right).
\]
An instance $I$ of $sch(\Sigma)$ is a stable model of $\Sigma$ if it is a (classical) model of $SM[\Sigma]$; note that here we adopt the unique-name assumption, that is, every constant of $C$ refers to a distinct object. We write $sms(\Sigma)$ for the set of stable models of $\Sigma$. Finally, given a database $D$ of $sch(\Sigma)$, an instance $I$ of $sch(\Sigma)$ is a stable model of $D$ and $\Sigma$ if $I \in sms(\Sigma[D])$ with $2[D] = \{ \text{True} \rightarrow a \mid a \in D \} \cup \Sigma$. We write $sms(D, \Sigma)$ for the set of stable models of $D$ and $\Sigma$, i.e., $sms(D, \Sigma) = sms(\Sigma[D])$.

Probability Spaces. A discrete probability space is a pair $(\Omega, P)$, where $\Omega$ is a finite or countably infinite set, called the sample space, and $P$ is a function $\Omega \rightarrow \{0, 1\} \cup \{\text{null}\}$, called discrete probability distribution over $\Omega$. Let $P_{\Omega}$ be the set that collects all the discrete probability distributions over $\Omega$. A parameterized probability distribution over $\Omega$ is a function $\delta : \mathbb{R}^k \rightarrow P_{\Omega}$ with $k > 0$ being the parameter dimension, i.e., $\delta(\cdot)$ is a discrete probability distribution over $\Omega$ for every parameter instantiation $\bar{p} \in \mathbb{R}^k$. For the sake of presentation, we write $\delta(\bar{p})$ instead of $\delta(\cdot)$ to avoid the overuse of parentheses, i.e., expressions of the form $\delta(\cdot)(\cdot)$. Due to our assumption that constants are essentially real numbers, it suffices to consider discrete probability distributions that are numerical, namely over a sample space $\Omega \subseteq \mathbb{R}$. Here is a simple example that illustrates the notion of parameterized probability distribution.

Example 3.1 (Parameterized Probability Distribution). The throwing of an unbiased die can be modelled via the discrete probability space $(\Omega, P)$, where $\Omega = \{1, \ldots, 6\}$, the six faces of a die, and $P$ is such that $P(i) = \frac{1}{6}$ for each $i \in \Omega$. On the other hand, we can model the throwing of a biased die by using a parameterized probability distribution. In particular, this can be done via $Die : \mathbb{R}^5 \rightarrow P_{\Omega}$ with $\Omega = \{0, 1, \ldots, 6\}$ defined as follows: for every $\bar{p} = (p_1, \ldots, p_6) \in \mathbb{R}^5$, $\delta(\bar{p})(i) = 0$ if $p_i = 0$ and $\delta(\bar{p})(i) = p_i$ for each $i \in [6]$, otherwise, $\delta(\bar{p})(i) = 1$ and $\delta(\bar{p})(i) = 0$ for each $i \in [6]$. Note that the outcome 0 is associated with incorrect instantiations of the parameters.

Although the notion of discrete probability space is very useful for our work, it does not suffice since we are going to encounter uncountable sample spaces. Thus, we also need the notion of probability space that goes beyond countable sample spaces. Let $\Omega$ be a (possibly uncountable) set. A $\sigma$-algebra over $\Omega$ is a subset $\mathcal{F}$ of $2^\Omega$ (the powerset of $\Omega$), i.e., a collection of subsets of $\Omega$, that (i) contains $\Omega$, (ii) is closed under complement, i.e., $\mathcal{F} \cap \mathcal{F} = \emptyset$, and (iii) is closed under countable unions, i.e., for a countable set $\mathcal{E}$ of elements of $\mathcal{F}$, $\bigcup_{E \in \mathcal{E}} E \in \mathcal{F}$. As a consequence, a $\sigma$-algebra contains the empty set and is closed under countable intersections.

A probability space is defined as a triple $(\Omega, \mathcal{F}, P)$, where
- $\Omega$ is a (possibly uncountable) set, called the sample space,
- $\mathcal{F}$ is a $\sigma$-algebra over $\Omega$, called the event space, and
- $P : \mathcal{F} \rightarrow [0, 1]$, called a probability measure, is such that $P(\Omega) = 1$, and, for every countable set $\mathcal{E}$ of pairwise disjoint elements of $\mathcal{F}$, $P(\bigcup_{E \in \mathcal{E}} E) = \sum_{E \in \mathcal{E}} P(E)$.

For a non-empty $\mathcal{F} \subseteq 2^\Omega$, the closure of $\mathcal{F}$ under complement and countable unions is a $\sigma$-algebra and it is said to be generated by $\mathcal{F}$.

4 GENERATIVE DATALOG WITH NEGATION

A Datalog\(^\neg\) program (which is essentially a TGD\(^\neg\) program without existentially quantified variables) specifies how to obtain a set of stable models from an input database. In this section, we present
generative Datalog\(^{\gamma}\) programs that essentially specify how to infer a distribution over sets of stable models from an input database. We first present the syntax of generative Datalog\(^{\gamma}\) programs. We then give an informal description of their semantics, highlighting the challenges in defining the probabilistic semantics in question, and then proceed with the formal definition. Henceforth, by \(\Delta\) we refer to a finite set of parameterized numerical probability distributions.

**Syntax.** A \(\Delta\)-term is an expression of the form \(\delta(p)[q]\), where \(p\) is a non-empty tuple of terms, \(q\) is a (possibly empty) tuple of terms, and \(\delta\) belongs to \(\Delta\) with parameter dimension \([p]\). We call \(p\) the distribution parameters and \(q\) the (optional) event signature. When the event signature is missing we simply write \(\delta(p)\). Intuitively, \(\delta(p)[q]\) denotes a sample from the probability distribution \(\delta(p)\), where different samples are drawn for different event signatures. A \(\Delta\)-atom over a schema \(S\) is an expression of the form \(R(t_1, \ldots, t_n)\), where \(R \in S\), \(ar(R) = n\), and \(t_i\) is either an ordinary term (constant or variable), or a \(\Delta\)-term for each \(i \in [n]\). In other words, a \(\Delta\)-atom is an atom that can mention, apart from terms, also \(\Delta\)-terms.

A generative Datalog rule with negation w.r.t. \(\Delta\) (or simply GDatalog\(^{\gamma}\) program \(\Pi\) over a schema \(S\) is an expression of the form

\[
R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n) \rightarrow \neg P_1(\bar{v}_1), \ldots, \neg P_m(\bar{v}_m) \rightarrow R_0(\bar{w})
\]

for \(n, m \geq 0\), where \(R_0, R_1, \ldots, R_n, P_1, \ldots, P_m\) are predicates of \(S\), \(\bar{u}_1, \ldots, \bar{u}_n\) are tuples over \(C \cup V\), \(\bar{v}_1, \ldots, \bar{v}_m\) are tuples over \(C \cup V\) such that each of their variables is mentioned in \(\bar{u}_j\) for some \(j \in [n]\), and \(\bar{w}\) is a tuple consisting of terms from \(C \cup V\) and \(\Delta\)-terms such that each of its variables, possibly as part of the distribution parameters or the event signature of a \(\Delta\)-term, is mentioned in \(\bar{u}_j\) for some \(j \in [n]\). The \(\Delta\)-atom that appears on the right of the right symbol is the head of \(\rho\), denoted \(H(\rho)\), while the expression on the left of the right symbol is the body of \(\rho\), denoted \(B(\rho)\). We further write \(B^+(\rho)\) for the set of positive literals in \(B(\rho)\) and \(B^-\) for the set of atoms appearing in the negative literals of \(B(\rho)\).

A GDatalog\(^{\gamma}\)\(\{\Delta\}\) program \(\Pi\) over \(S\) is a finite set of GDatalog\(^{\gamma}\) rules over \(S\). A predicate \(R \in S\) occurring in \(\Pi\) is called extensional if there is no rule of the form body \(\rightarrow R(\bar{w})\) in \(\Pi\), i.e., \(R\) occurs only in rule-bodies; otherwise, it is called intensional. The intensional (database) schema of \(\Pi\), denoted \(edb(\Pi)\), consists of the extensional predicates in \(\Pi\), while the extensional schema of \(\Pi\), denoted \(idb(\Pi)\), is the set of all intensional predicates in \(\Pi\). The schema of \(\Pi\), denoted \(sch(\Pi)\), is the set \(edb(\Pi) \cup idb(\Pi)\). Given a GDatalog\(^{\gamma}\)\(\{\Delta\}\) program \(\Pi\) and a database \(D\) of \(edb(\Pi)\), \(\Pi[D]\) is the program \(\{ \rightarrow a \mid a \in D \} \cup \Pi\).

*Example 4.1.* Consider the “network resilience” example already discussed in the Introduction. Recall that the network is dominated by the malware if all routers are infected or isolated (i.e., connected only to infected routers). Assume we have the following schema:

\[
\text{Router}(\text{router\_id}) \quad \text{Infected}(\text{router\_id}, \text{boolean}) \quad \text{Uninfected}(\text{router\_id}).
\]

Consider the parameterized distribution \(\text{Flip} : \mathbb{R} \rightarrow P([0,1])\) over \([0,1]\) such that \(\text{Flip}(\rho) = p\) and \(\text{Flip}(\rho') = 1 - p\), and let \(\Delta = \\{\text{Flip}\}\). The GDatalog\(^{\gamma}\)\(\{\Delta\}\) program \(\Pi\) over the above schema that encodes the malware domination cases is as follows:

\[
\text{Infected}(x, 1), \text{Connected}(x, y) \rightarrow \text{Infected}(y, \text{Flip}(0.1)[x, y])
\]

\[
\text{Router}(x), \neg\text{Infected}(x, 1) \rightarrow \text{Uninfected}(x)
\]

\[
\text{Uninfected}(x), \neg\text{Uninfected}(y), \text{Connected}(x, y) \rightarrow \bot,
\]

where \(\bot\) denotes False. The symbol \(\bot\) is syntactic sugar as False can be always simulated using stable negation. In particular, we can replace \(\bot\) with a 0-ary predicate \(\text{Fail}\) that is forced to be false in every stable model via the rule \(\neg\text{Aux} \rightarrow \text{Aux}\), where \(\text{Aux}\) is a new predicate not mentioned in the program in question.

**An Informal Semantics.** Consider a positive (i.e., without negation) GDatalog\(^{\gamma}\)\(\{\Delta\}\) program \(\Pi\). Every configuration (i.e., a set) \(C\) of probabilistic choices made via the \(\Delta\)-terms during the execution of \(\Pi\) on an input database \(D\) leads to a different minimal model \(M_C\) of \(D\) and \(\Pi\) (in fact, of \(D\) and a set of TGDs \(\Sigma\) that somehow captures the intended meaning of \(\Pi\)). Furthermore, the probabilistic choices of \(C\) are explicitly represented in \(M_C\), which means that \(M_C\) carries enough information that allows us to calculate its probability. According to the probabilistic semantics from [3], the output of \(\Pi\) on \(D\) is a probability space over the sample space consisting of all the minimal models of \(D\) and \(\Sigma\) with non-zero probability, the so-called possible outcomes. This is a natural generalization of the (deterministic) semantics of Datalog, where the output of a positive program on an input database is their unique minimal model.

Once we add negation to generative Datalog, one may be tempted to think that we can follow a similar approach by considering stable instead of minimal models. However, in the presence of stable negation, the situation changes significantly. A configuration of probabilistic choices no longer leads to a single minimal model, but rather a (possibly empty) set of stable models. Consider the GDatalog\(^{\gamma}\)\(\{\{\text{Flip}\}\}\) program \(\Pi_{\text{coin}}\) consisting of

\[
\rightarrow \text{Coin}(\text{Flip}(0.5)) \quad \text{Coin}(1), \neg\text{Aux}_1 \rightarrow \text{Aux}_2
\]

\[
\text{Coin}(0) \rightarrow \bot \quad \text{Coin}(1), \neg\text{Aux}_2 \rightarrow \text{Aux}_1
\]

which encodes the flipping of a “fair” coin. Intuitively, if the flipping result to 0 which corresponds to \(\text{Heads}\), then there is no stable model, but if the result is 1 (which corresponds to \(\text{Tails}\), then \(\{\text{Aux}_1, \text{Coin}(1)\}\) and \(\{\text{Aux}_2, \text{Coin}(1)\}\) are the stable models of \(\Pi_{\text{coin}}\) (in fact, similarly to the positive case, of a TGD \(\Sigma_{\text{coin}}\) that somehow captures the intended meaning of \(\Pi_{\text{coin}}\); here the database is empty). It may also happen that different configurations of probabilistic choices lead to the same set of stable models (for example, if we add to \(\Pi_{\text{coin}}\) the rule \(\text{Coin}(1) \rightarrow \bot\)).

The above informal discussion essentially tells us that, in general, there is no correspondence between stable models and configurations of probabilistic choices. Thus, it will be conceptually problematic to directly consider individual stable models as possible outcomes, which in turn form the underlying sample space. This leads to the obvious idea of considering sets of stable models (including the empty set) with non-zero probability, obtained via certain configurations of probabilistic choices, as possible outcomes. Coming back to the “coin” example, this would result in the possible outcome \(\{\{\text{Aux}_1, \text{Coin}(1)\}, \{\text{Aux}_2, \text{Coin}(1)\}\}\) for the configuration where the result of the flip is 1, and the possible outcome 0 for the configuration where the result of the flip is 0, and the two respective events should have probability 0.5 each.
Although the above idea is a plausible one, which we have thoroughly explored, it brings us to the other complication that is coming with negation. Unlike positive generative Datalog, the probabilistic choices are not necessarily reflected in the resulting set of stable models. In other words, there are cases where it is not possible to extract the configuration of probabilistic choices that led to a set of stable models \( M \) by simply inspecting \( M \). This can be seen in the “coin example”, where the fact that the result of the flip is 0 is not explicitly encoded in the respective stable model, that is, the empty model. This is problematic as a set of stable models does not carry enough information that allows us to calculate its probability. To overcome this complication, the key idea is to symbolically represent a set consisting of the following TGD \( \rho \) quantified variables obtained after converting program. Consider a GDatalog configuration of probabilistic choices may have a finite or an infinite possible outcomes as invalid ones, but rather (infinitary) ground programs that lead to sets of stable models. There is, however, a crucial choice to be made here: infinite possible outcomes as invalid ones, Grohe et al. in [16], where generative Datalog with continuous distributions is studied, it is both technically and conceptually more meaningful to consider infinite possible outcomes as invalid ones, which are collected in the so-called error event. In our development, we adopt the view by Grohe et al. [16].

This brings us to the importance of the adopted grounding strategy since, depending on how sophisticated this strategy is, a certain configuration of probabilistic choices may have a finite or an infinite representative grounding, which in turn affects the resulting probability space. Thus, the definition of our semantics is parameterized by a grounder, and we will discuss a mechanism for performing a comparison among semantics induced by different grounders.

**From GDatalog\(^\sim\)[\( \Delta \)] to TGD\(^\sim\).** We proceed to formalize the translation of a generative Datalog program with negation into a TGD\(^\sim\) program. Consider a GDatalog\(^\sim\)[\( \Delta \)] rule \( r \):

\[
R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), \neg P_{r_1}(\bar{v}_1), \ldots, \neg P_{m}(\bar{v}_m) \rightarrow R_0(\bar{w}),
\]

where \( \bar{w} = (w_1, \ldots, w_{ar(R_0)}) \) with \( w_i = \delta_1(p_1)[\bar{q}_1], \ldots, w_{ir} = \delta_r(p_r)[\bar{q}_r] \), for \( 1 \leq i_1 < \cdots < i_r \leq ar(R_0) \), be all the \( \Lambda \)-terms in \( \bar{w} \). If \( r = 0 \) (i.e., there are no \( \Lambda \)-terms in \( \bar{w} \)), then \( \rho_3 \) is defined as the singleton set \( \{ \Sigma_\rho \} \), where \( \Sigma_\rho \) is the TGD\(^\sim\) without existentially quantified variables obtained after converting \( r \) into a first-order sentence in the usual way: commas are treated as conjunctions and all the variables are universally quantified. Otherwise, \( \rho_3 \) is defined as the set consisting of the following TGD\(^\sim\):

\[
R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n), \neg P_{r_1}(\bar{v}_1), \ldots, \neg P_{m}(\bar{v}_m), \rightarrow \text{Active}_{\delta_j}^{\phi_j}(\bar{p}_j, \bar{q}_j) \quad \text{for each } j \in [r],
\]

\[
\text{Active}_{\delta_j}^{\phi_j}(\bar{p}_j, \bar{q}_j) \rightarrow \exists y_j \text{ Result}_{\delta_j}^{\phi_j}(\bar{p}_j, \bar{q}_j, y_j)
\]

where Active\(^{\phi_j}_{\delta_j}\)(\( \bar{p}_j, \bar{q}_j \)) and Result\(^{\phi_j}_{\delta_j}\)(\( \bar{p}_j, \bar{q}_j \)) are fresh \((|\bar{p}_j| + |\bar{q}_j|)\)-ary and \((|\bar{p}_j| + |\bar{q}_j| + 1)\)-ary predicates, respectively, not occurring in sch(\( \Pi \)), \( y_1, \ldots, y_r \) are distinct variables not occurring in \( r \), and \( \bar{w}' = (w_{i_1}, \ldots, w_{i_1}, \bar{q}_1, \ldots, w_{i_r}, \bar{q}_r, \ldots, w_{ir}, \bar{q}_r) \) is obtained from \( \bar{w} \) by replacing \( \Lambda \)-terms with variables \( y_1, \ldots, y_r \).

We proceed to formalize the translation of a generative Datalog program into a TGD\(^\sim\) program. Consider a GDatalog\(^\sim\)[\( \Delta \)] program \( \Pi \) of the form \( \text{Active}_{\phi}^{\delta}(\bar{p}, \bar{q}) \) and \( \text{Result}_{\phi}^{\delta}(\bar{p}, \bar{q}, o) \) for every two ground AtR TGDs \( \sigma_\phi \) and \( \sigma_{\phi'} \) of the form

\[
\alpha \rightarrow \text{Active}_{\delta}^{\phi}(\bar{p}, \bar{q}, o) \quad \text{and} \quad \alpha \rightarrow \text{Result}_{\delta}^{\phi}(\bar{p}, \bar{q}, o', \bar{o})
\]

respectively, it holds that \( o = o' \). Intuitively, this means that \( \Sigma \) encodes valid probabilistic choices. We write \( [\text{ground}(\Sigma_\Delta)]_1 \) for all the consistent subsets of \( \text{ground}(\Sigma_\Delta)_1 \). Observe that a consistent subset \( \Sigma \) of \( \text{ground}(\Sigma_\Delta)_1 \) induces a partial function \( \text{AtR}_{\Sigma} : \text{Act} \rightarrow \text{Res} \), where \( \text{Act} \) (resp., \( \text{Res} \)) is the set of all atoms that can be formed using predicates of the form \( \text{Active}^{\phi}_{\delta}(\bar{p}, \bar{q}) \) (resp., \( \text{Result}^{\phi}_{\delta}(\bar{p}, \bar{q}, o') \)) of sch(\( \Sigma \)), and constants from \( \Sigma \). We say that \( \text{AtR}_{\Sigma} \) is compatible with a set \( \Sigma' \subseteq \text{ground}(\Sigma_\Delta)_1 \) if, written \( \text{AtR}_{\Sigma} \rightarrow \Sigma' \), if it is defined on every atom of the form \( \text{Active}_{\phi}^{\delta}(\bar{p}, \bar{q}) \) occurring in heads(\( \Sigma' \)), that is, the set of...
we further define the sets $\Sigma' \in [2^{\text{ground}(\Sigma)}]_\|=\Sigma$, such that $\Sigma \subseteq \Sigma'$ and $\text{AtR}_\Sigma$ is total. We can now introduce the desired notion of grounding.

**Definition 4.3 (Grounding Generative Datalog**). Consider a GDatalog $[\Lambda]$ program $\Pi$. A grounder of $\Pi$ is a monotonic function $G : [2^{\text{ground}(\Sigma)}]_\|=\Sigma$, such that, for every $\Sigma \in [2^{\text{ground}(\Sigma)}]_\|=\Sigma$, if $\text{AtR}_\Sigma \rightarrow G(\Sigma)$, then $\text{sms}(G(\Sigma) \cup \Sigma) = \text{sms}(\Sigma) \cup \Sigma'$ for every totalizer $\Sigma'$ of $\text{AtR}_\Sigma$. The G-grounding of $\Pi$, denoted $G\text{-ground}(\Pi)$, is defined as the set $\{\Sigma \cup G(\Sigma) \mid \Sigma \in \text{terminals}(G) \text{ and there is no } \Sigma' \in \text{terminals}(G) \text{ such that } \Sigma' \subseteq \Sigma\}$ with terminals$(G) = \{\Sigma \in [2^{\text{ground}(\Sigma)}]_\|=\Sigma, \text{AtR}_\Sigma \rightarrow G(\Sigma)\}$. The importance of the monotonicity of grounders for generative Datalog is revealed in Section 5, where we present our chase-based semantics. Before proceeding with our probabilistic semantics, we present a concrete example of a grounder, which is conceptually very simple (hence the name “simple grounder”), but at the same time a relevant one as we shall see later in the paper.

**Simple Grounder.** Given a GDatalog $[\Lambda]$ program $\Pi$, we define the so-called simple grounder $G_{\text{Simple}}$, which relies on an operator that operates on (infinitary) existential-free TGD programs. Consider an (infinitary) TGD program $\Sigma$ with existentially quantified variables. We define the operator $G_{\text{Simple}}$, that extends an (infinitary) TGD program $\Sigma'$ in $2^{\text{ground}(\Sigma)}$ by matching the positive body literals of $\Sigma$ with ground head atoms of $\Sigma'$. The notion of matching can be formalized via homomorphisms with which we assume the reader is familiar. We write $h(A) \subseteq B$ to indicate that $h$ is a homomorphism from a set of atoms $A$ to a set of atoms $B$. Given a TGD $\Sigma$ and a homomorphism $h$ from $B^+(\sigma)$ to some other set of atoms, we write $h(\sigma)$ for the TGD obtained by applying $h$ to the body and head of $\sigma$. We define $G_{\text{Simple}}$, the monotonic function from $2^{\text{ground}(\Sigma)}$ to $2^{\text{ground}(\Sigma')}$ such that, for every $\Sigma' \in 2^{\text{ground}(\Sigma)}$, $G_{\text{Simple}}(\Sigma') = \Sigma' \cup \{h(\sigma) \mid \sigma \in \Sigma \text{ and } h(B^+(\sigma)) \subseteq \text{heads}(\Sigma')\}$.

We further define the sets $G_{\text{Simple}}^0(\Sigma') = \Sigma'$,

$G_{\text{Simple}}^{i+1}(\Sigma') = G_{\text{Simple}}(G_{\text{Simple}}^i(\Sigma'))$ for $i > 0$,

and let $G_{\text{Simple}}^\infty(\Sigma') = \bigcup_{i \geq 0} G_{\text{Simple}}^i(\Sigma')$.

Due to the monotonicity of $G_{\text{Simple}}$, it is clear that $G_{\text{Simple}}^\infty(\Sigma')$ is the least fixpoint of $G_{\text{Simple}}$ that contains $\Sigma' \in 2^{\text{ground}(\Sigma)}$. We are now ready to define the function $G_{\text{Simple}}$.

**Definition 4.4 (Simple Grounder).** Consider a GDatalog $[\Lambda]$ program $\Pi$. $G_{\text{Simple}} : [2^{\text{ground}(\Sigma)}]_\|=\Sigma$, such that $G_{\text{Simple}}(\Sigma) = G_{\text{Simple}}^\infty(\emptyset) \cup \Sigma$ where $\Sigma' = \Sigma_\Pi \cup \Sigma$, for every $\Sigma \in [2^{\text{ground}(\Sigma)}]_\|=\Sigma$.

It is not difficult to show that $G_{\text{Simple}}$ is indeed a grounder:

**Proposition 4.5.** Consider a GDatalog $[\Lambda]$ program $\Pi$, it holds that $G_{\text{Simple}}$ is a grounder of $\Pi$.

An example that illustrates the simple grounder follows:

**Example 4.6.** Let $\Pi$ be the GDatalog $[\Lambda]$ program from Example 4.1. Its translation into the TGD $[\Lambda]$ program $\Sigma$ can be found in Example 4.2. Consider the database $D$ defined as $\{\text{Connected}(i, j) \mid i, j \in [3] \text{ and } i \neq j\}$ that stores a fully connected network consisting of three routers, the first being initially infected. Hence, $G_{\text{Simple}}(\emptyset)$ contains

$\{\text{Infected}(1, 1), \text{Connected}(1, 2) \rightarrow \text{Active}_{\text{Flip}}(0, 1, 1, 2)\}$

$\{\text{Connected}(1, 1), \text{Connected}(1, 3) \rightarrow \text{Active}_{\text{Flip}}(0, 1, 1, 3)\}$

and the following TGD $\Delta$ for all $i, j \in [3]$ with $i \neq j$:

$\text{Router}(i), \neg\text{Infected}(i, 1) \rightarrow \text{Uninfected}(i)$

$\text{Uninfected}(i), \neg\text{Infected}(i) \rightarrow \text{Connected}(i, j) \rightarrow \bot$.

Now, considering the program

$\Sigma = \{\text{Active}_{\text{Flip}}(0, 1, 1, i) \rightarrow \text{Result}_{\text{Flip}}(0, 1, 1, i, 0) \mid i \in \{2, 3\}\}$,

$G_{\text{Simple}}(\emptyset)$ extends $G_{\text{Simple}}(\emptyset)$ with

$\text{Connected}(1, 1), \text{Connected}(1, 3) \rightarrow \text{Active}_{\text{Flip}}(0, 1, 1, 2)$

$\text{Result}_{\text{Flip}}(0, 1, 1, i, 0), \text{Infected}(1, 1), \text{Connected}(1, i) \rightarrow

\text{Infected}(i, 0)$

for each $i \in \{2, 3\}$. We have that $\Sigma \in \text{terminals}(G_{\text{Simple}}(\emptyset))$ and $\Sigma \cup G_{\text{Simple}}(\emptyset) \in G_{\text{Simple}}(\emptyset)$.

**Probabilistic Semantics.** We now have all the ingredients for defining the semantics of generative Datalog. We start by formalizing the notion of possible outcomes, which in turn gives rise to the sample space over which we need to define a probability space.

**Definition 4.7 (Possible Outcome).** Consider a GDatalog $[\Lambda]$ program $\Pi$ and a database $D$ of edb(\Pi). Let $G$ be a grounder of $\Pi[D]$. A possible outcome of $D$ w.r.t $G$ is a (possibly infinitary) program $\Sigma \in G\text{-ground}(\Pi[D])$ and $\text{sms}(\Sigma) \cup \text{sms}(\Pi[D])$ such that $\delta(\sigma)(\sigma)$ $> 0$ for every $\Lambda$-atom $\text{Result}_i(q, q, q, o) \in \text{heads}(\Sigma)$. We write $\Omega_{G}(D)$ for the set of all possible outcomes of $D$ w.r.t $G$.

Observe that each $\Sigma \in \Omega_{G}(D)$ induces a (possibly empty) sub-set of $\text{sms}(\Pi[D])$, that is, $\text{sms}(\Pi[D])$. Note that, for $\Sigma' \neq \Sigma, \Sigma' \in \Omega_{G}(D)$, $\Sigma' \neq \Sigma$ does not imply $\text{sms}(\Sigma') \neq \text{sms}(\Sigma)$, i.e., two different possible outcomes may induce the same subset of $\text{sms}(\Pi[D])$.

Given a GDatalog $[\Lambda]$ program $\Pi$ and a database $D$ of edb(\Pi), the output of $\Pi$ on $D$, relative to some grounder $G$ of $\Pi[D]$, should be understood as a probability space over $\Omega_{G}(D)$. We proceed to explain how such a probability space can be defined. In essence, the events that we are interested in are subsets of $\text{sms}(\Pi[D])$ induced by proper (i.e., finite) programs of $\Omega_{G}(D)$, or, in other words, by finite possible outcomes of $D$ w.r.t $G$; we write $\Omega_{G}(D)$ for the set that collects all the (finite) programs of $\Omega_{G}(D)$. Therefore, finite possible outcomes resulting in the
same set of stable models of $\text{sms}(\Sigma_{\Pi[D]})$ must belong to same event. Moreover, following Grohe et al. on generative Datalog with continuous distributions [16], we collect all the infinite possible outcomes of $\Omega_{\Pi[G]}(D)$ in the infinity (or error) event $\Omega_{\Pi[G]}^\infty(D)$.

Let $P_{\Pi[G]}^D$ be the subset of $\Omega_{\Pi[G]}^{\Omega}(D)$ consisting of $\Omega_{\Pi[G]}^{\Omega}(D)$ and all the maximal subsets $E$ of $\Omega_{\Pi[G]}^{\Omega}(D)$ such that, for all sets $\Sigma', \Sigma'' \in E$, $\text{sms}(\Sigma') = \text{sms}(\Sigma'')$. We define $F_{\Pi[G]}^D$ as the $\sigma$-algebra over $\Omega_{\Pi[G]}(D)$ generated by $F_{\Pi[G]}^D$. We finally define the function $P_{\Pi[G]}^D : F_{\Pi[G]}^D \rightarrow [0, 1]$ as follows. For every $\Sigma \in \Omega_{\Pi[G]}^{\Omega}(D)$, let

$$Pr(\Sigma) = \prod_{\delta(\Sigma)}\delta(\hat{p}(a)).$$

Then, for every (countable) set $E \in F_{\Pi[G]}^D$, with $E \subseteq \Omega_{\Pi[G]}^{\Omega}(D)$, let

$$P_{\Pi[G]}^D(E) = \sum_{\Sigma \in E} Pr(\Sigma).$$

Observe that $\Omega_{\Pi[G]}^{\Omega}(D) \subseteq F_{\Pi[G]}^D$, and thus, we can let

$$P_{\Pi[G]}^D(\Omega_{\Pi[G]}^{\Omega}(D)) = 1 - P_{\Pi[G]}^D(\Omega_{\Pi[G]}^{\Omega}(D)).$$

Finally, $P_{\Pi[G]}^D$ naturally extends to countable unions via countable addition. We are now ready to define the output of a generative Datalog program on an input database relative to a grounder.

**Definition 4.8 (Output of GDatalog “[\Delta]” Programs).** Consider a GDatalog “[\Delta]” program $\Pi$ and a database $D$ of edb($\Pi$). Let $G$ be a grounder of $\Pi[D]$. The output of $\Pi$ on $D$ relative to $G$, denoted $\Pi[G](D)$, is defined as the triple $(\Omega_{\Pi[G]}(D), F_{\Pi[G]}^D, P_{\Pi[G]}^D).$

It can be shown that the following holds, which provides the desired probabilistic semantics for generative Datalog$^\Omega$.

**Theorem 4.9.** Consider a GDatalog$^\Omega”$[\Delta]” program $\Pi$, a database $D$ of edb($\Pi$), and a grounder $G$ of $\Pi[D]$. $\Pi[G](D)$ is a probability space.

We proceed to show the semantics for generative Datalog$^\Omega$ in action by exploiting our “network resilience” example.

**Example 4.10.** Continuing Example 4.6, $\Sigma \in \Omega_{\Pi[G,Simple\Pi[D]}}^{\Omega}(D)$ and $Pr(\Sigma) = \text{Flip}(0.1)(0)^2 = 0.09$. It can be verified that all other possible outcomes are finite and induce a non-empty subset of $\text{sms}(\Sigma[D])$. This implies that $\Sigma \in F_{\Pi[G,Simple\Pi[D]}}^D$, and the probability of the event $\Pi[D]$ has some stable model $\pi$ is

$$P_{\Pi[G,Simple\Pi[D]]}^D(\Omega_{\Pi[G,Simple\Pi[D]]}^{\Omega}(D) \setminus \{\Sigma\}) = 1 - P_{\Pi[G,Simple\Pi[D]]}^D(\{\Sigma\}) = 1 - 0.09^2 = 0.19.$$

Summing up, the network stored in the database $D$ (given in Example 4.6) is dominated by the malware with probability 0.19.

**Positive Programs.** A natural question is whether the probabilistic semantics proposed above for generative Datalog$^\Omega$ coincides with the existing semantics for generative Datalog introduced in [3] if we concentrate on positive programs that guarantee that all the possible outcomes are finite, and thus, the error event takes probability zero. It should not be overlooked, however, that the probabilistic semantics for generative Datalog$^\Omega$ is actually a family of semantics since, depending on the adopted grounder, we get different semantics; a mechanism for performing a qualitative comparison among the different semantics is discussed below. Therefore, the key question is whether there exists a grounder for generative Datalog$^\Omega$ that gives rise to semantics that coincides with that of [3]. Interestingly, we can show that such a grounder exists; in fact, we can show that this is the simple grounder introduced above (see Definition 4.4).

**Qualitative Comparison of Probabilistic Semantics.** As already discussed, Theorem 4.9 gives rise to a family of semantics for generative Datalog$^\Omega$ since different grounders lead to different semantics. It is thus natural to ask how we can qualitatively compare the various semantics. Recall that infinite possible outcomes are considered as invalid ones that are collected in the error event, and, in general, the larger the error event, the less the probability assigned to valid (that is, finite) possible outcomes. This essentially tells us that the greater the probability assigned to finite outcomes by a probability measure, the better. Recall that the output of a program $\Pi$ on a database $D$ relative to a grounder $G$ of $\Pi[D]$ is the probability space $(\Omega_{\Pi[G]}(D), F_{\Pi[G]}^D, P_{\Pi[G]}^D)$.

**Definition 4.11 (Comparison of Semantics).** Consider a GDatalog$^\Omega”$[\Delta]” program $\Pi$ and a database $D$ of edb($\Pi$). Let $G, G'$ be grounders of $\Pi[D]$. We say that $\Pi[G](D)$ is as good as $\Pi[G'](D)$ if

$$P_{\Pi[G]}^D(\{\Sigma \in \Omega_{\Pi[G]}^{\Omega}(D) | \text{sms}(\Sigma) = I\}) \geq P_{\Pi[G']}^D(\{\Sigma \in \Omega_{\Pi[G']}^{\Omega}(D) | \text{sms}(\Sigma) = I\})$$

for every set of stable models $I \subseteq \text{sms}(\Sigma[D])$.

The above notion equips us with a mechanism that can guide the choice of the probabilistic semantics based on the class of generative Datalog$^\Omega$ that we are interested in, which in turn depends on the underlying application. For example, if we are interested only in positive generative Datalog programs, we can then show that there is no need to go beyond the probabilistic semantics induced by the simple grounder, i.e., the semantics induced by the simple grounder is as good as the semantics induced by any other grounder.

**Theorem 4.12.** Consider a GDatalog$^\Omega”$[\Delta]” program $\Pi$ and a database $D$ of edb($\Pi$). Let $G$ be an arbitrary grounder of $\Pi[D]$. It holds that $\Pi[G,Simple\Pi[D]](D)$ is as good as $\Pi[G](D)$.

**5 FIXPOINT PROBABILISTIC SEMANTICS**

We now tackle the question whether the probabilistic semantics presented in the previous section can be equivalently defined via a fixpoint procedure. An affirmative answer to this question will provide a procedure that is amenable to practical implementations. To this end, we introduce a novel chase procedure for generative Datalog$^\Omega$, which we then use to define a chase-based probability space that leads to the desired fixpoint probabilistic semantics.

**Chasing Generative Datalog$^\Omega$ Programs.** We start by introducing our novel chase procedure. Let us stress that our chase procedure deviates from the standard one, which typically operates over databases with the aim of completing an incomplete database as dictated by a given set of TGDs. Our chase procedure operates on ground AtR TGDs with the aim of completing such programs
as dictated by a given ground TGDD program. We start with the notion of trigger application, which corresponds to a chase step.

**Definition 5.1 (Trigger Application).** Consider a GDatalog\(^{\Delta}\) program \(\Pi\), and let \(\Sigma \in \mathcal{G}^{\text{ground}}(X^2)\) and \(\Sigma' \in \mathcal{G}^{\text{ground}}(X^2)\). A trigger for \(\Sigma'\) on \(\Sigma\) is an atom \(\alpha = \text{Active}_{\Sigma'}(\bar{p}, \bar{q}) \in \text{heads}(\Sigma')\) such that there is no TGD in \(\Sigma\) of the form \(\alpha \rightarrow \text{Return}_{\Sigma}(\bar{q}, \bar{p}, o)\) for some \(o \in C\). An application of \(\alpha\) to \(\Sigma\) returns the set of (infinitary) TGD programs \(\{\Sigma_1, \Sigma_2, \ldots\}\) such that the following hold:

- for each integer \(i > 0\), \(\Sigma_i = \Sigma \cup \{\alpha \rightarrow \text{Return}_{\Sigma}(\bar{q}, \bar{p}, o)\}\), where \(o \in C\) and \(\delta(\bar{p})(o) > 0\), and
- for each constant \(o \in C\) with \(\delta(\bar{p})(o) > 0\), there exists \(i > 0\) such that \(\Sigma_i = \Sigma \cup \{\alpha \rightarrow \text{Return}_{\Sigma}(\bar{q}, \bar{p}, o)\}\).

Such a trigger application is denoted as \(\Sigma(\alpha)\).

Having the notion of trigger application, we can now introduce the central notion of chase tree, which is relative to a grounder.

**Definition 5.2 (Chase Tree).** Consider a GDatalog\(^{\Delta}\) program \(\Pi\) and a database \(D\) of edb(\(\Pi\)). Let \(G\) be a grounder of \(\Pi(G)\). A chase tree for \(D\) w.r.t. \(G\) is a (possibly infinite) rooted labelled tree \(T = (N, E, \lambda)\), where \(N : N \rightarrow \mathcal{E}^{\text{ground}}(X^2)\), such that:
- the root node is labelled by \(\emptyset\),
- for each non-leaf node \(v\) (i.e., \(v\) has an outgoing edge) with children \(v_1, v_2, \ldots\), there exists a trigger \(\alpha\) for \(G(\lambda(v))\) on \(\lambda(v)\) such that \(\lambda(v)(\alpha) = \lambda(v_1)(\lambda(v_2), \ldots)\), and
- for each leaf node \(v\) (i.e., \(v\) has no outgoing edges), there is no trigger for \(G(\lambda(v))\).

A chase tree \(T = (N, E, \lambda)\) essentially encodes iterative trigger applications. A maximal path of \(T\) is either a finite path \(v_1, \ldots, v_n\) in \(T\), where \(v_1\) is the root node and \(v_n\) is a leaf node, or an infinite path \(v_1, v_2, \ldots\) in \(T\), where \(v_1\) is the root node. Let \(\text{paths}(T)\) be the set of all maximal paths of \(T\); we further write \(\text{paths}^{\infty}(T)\) (resp., \(\text{paths}^{\ast}(T)\)) for the set of finite (resp., infinite) maximal paths of \(T\).

The result of a finite (resp., infinite) maximal path \(\pi = v_1, \ldots, v_n\) (resp., \(\pi = v_1, v_2, \ldots\)) of \(T\), denoted \(\Pi(\pi)\), is the program (resp., infinitary program) \(\lambda(v_n)(\lambda(v_{n-1}), \ldots)\). We further write \(\Pi(G)\) for \(\Pi(\lambda(\Pi(G)))\). The notations \(\Pi_{G}\) and \(\Pi_{G}\) naturally extend to sets of maximal paths, i.e., for a set \(M\) of maximal paths, \(\Pi(M) = \{\Pi(\pi) | \pi \in M\}\) such that \(\Pi(M) = \{\Pi(\pi) | \lambda(\Pi(G))\}\), and we may simply say chase tree relative to some grounder of \(\Pi(\Pi(G))\).

The next easy lemma collects some useful properties of chase trees.

**Lemma 5.3.** Consider a GDatalog\(^{\Delta}\) program \(\Pi\) and a database \(D\) of edb(\(\Pi\)). Let \(T = (N, E, \lambda)\) be a chase tree for \(D\) w.r.t. \(G\). Then:

1. For every \(v \in N\), \(\lambda(v) \in \mathcal{G}^{\text{ground}}(X^2)\).
2. For every \(u, v \in N\), \(u \neq v\) implies \(\lambda(u) \neq \lambda(v)\).

Item (1) essentially tells us that the nodes of a chase tree are labelled with sets of ground AR TGDs that are functionally consistent, while item (2) establishes an injectivity property. Another crucial property is the fact that two different chase trees, no matter in which order the triggers are applied, always lead to the same set of programs, that is, the set of programs obtained after collecting the results of their finite paths coincide. This is established by the next technical lemma, which exploits Lemma 5.3, as well as the monotonicity of grounders for generative GDatalog\(^{\Delta}\) programs.

**Lemma 5.4.** Consider a GDatalog\(^{\Delta}\) program \(\Pi\) and a database \(D\) of edb(\(\Pi\)). Let \(G\) be a grounder of \(\Pi(G)\). Then, for every \(T\) such that \(\Pi(T)\) is chase tree for \(D\) w.r.t. \(G\), it holds that

\[\text{paths}^{\infty}(T) = \text{paths}^{\ast}(T').\]

We finally establish the key connection between chase trees and possible outcomes, which heavily relies on Lemma 5.4.

**Lemma 5.5.** Consider a GDatalog\(^{\Delta}\) program \(\Pi\) and a database \(D\) of edb(\(\Pi\)). Let \(G\) be a grounder of \(\Pi(G)\). Then, for every \(T\) such that \(\Pi(T)\) is chase tree for \(D\) w.r.t. \(G\), the binary relation

\[\{\pi, \Pi(G) | \pi \in \text{paths}^{\infty}(T)\}\]

is a bijection from \(\text{paths}^{\infty}(T)\) to \(\Pi^{\infty}(\Pi(G))\).

**Fixpoint Probabilistic Semantics.** We now proceed to introduce the desired fixpoint semantics by exploiting the chase procedure presented above. Consider a GDatalog\(^{\Delta}\) program \(\Pi\) and a database \(D\) of edb(\(\Pi\)). Let \(G\) be a grounder of \(\Pi(G)\) and \(T\) be a chase tree for \(D\) w.r.t. \(G\). Our goal is to define a probability space \(PS_T\) over \(\text{paths}(T)\), and then show that from \(PS_T\) we can get a probability space that faithfully mimics \(\Pi(G)\).

We start by observing that each path \(\pi \in \text{paths}(T)\) induces a (possibly infinite) subset of \(\text{sms}(\Pi(G))\), that is, \(\text{sms}(\Pi(G))\). Note also that for \(\pi, \pi' \in \text{paths}(T)\), it might be the case that \(\pi \neq \pi'\) (and thus, by Lemma 5.3(2), \(\pi \neq \pi'\)), but they induce the same subset of \(\text{sms}(\Pi(G))\). We now proceed to define a probability space over \(\Omega_T = \text{paths}(T)\). Let \(F_T\) be the subset of \(2^{\text{paths}(T)}\) consisting of \(\text{paths}^{\infty}(T)\) and all the maximal subsets \(E\) of \(\text{paths}^{\ast}(T)\) such that, for all \(\pi, \pi' \in E\), \(\text{sms}(\Pi(G)) = \text{sms}(\Pi(G))\). Let \(F_T\) be the \(\sigma\)-algebra generated by \(F_T\). We finally define the function \(P_T : F_T \rightarrow [0, 1]\) as follows. For every \(\pi \in \text{paths}^{\infty}(T)\), let

\[P_T(\pi) = \prod_{\lambda(v_0)} \delta(\bar{p})(o),\]

where \(\lambda(v_0)\) is the root node and \(\lambda(n)\) is the leaves of \(T\), and \(\Pi(G)\). The notations \(\Pi(G)\) and \(\Pi(G)\) naturally extend to sets of maximal paths, i.e., for a set \(M\) of maximal paths, \(\Pi(G) = \{\Pi(\pi) | \pi \in M\}\) such that \(\Pi(G) = \{\Pi(\pi) | \lambda(\Pi(G))\}\), and we may simply say chase tree relative to some grounder of \(\Pi(\Pi(G))\).

Clearly, \(\text{paths}^{\infty}(T) \in F_T\), and thus, we can let

\[P_T(\text{paths}^{\infty}(T)) = 1 - P_T(\text{paths}^{\infty}(T)).\]

Finally, \(P_T\) extends to countable unions via countable addition. Clearly, \(P_S = (\Omega_T, F_T, P_T)\) is a probability space. Consider now the triple \(\Pi(PS_T)\) obtained from \(PS_T\) by replacing every \(\pi \in \Omega_T\) with \(\Pi(G)\). Formally, \(\Pi(PS_T) = (\Pi(G), F_T, P_T)\), where

- \(F_T = \{E | E \in F_T\}\),
- \(P_T(E) = \prod_{\Pi(G)} P_T(\Pi(G))\), where

We claim that \(\Pi(PS_T)\) is a probability space that faithfully mimics the probability space \(\Pi(G)\).

More precisely, by exploiting Lemma 5.5, we can show the following:
Consider a GDatalog-\* program \( \Pi \) and a database \( D \) of edb(\( \Pi \)). Let \( G \) be a grounder of \( \Pi[D] \) and \( T \) be a chase tree for \( D \) w.r.t. \( \Pi \) relative to \( G \). The following hold:

1. \( \| \text{paths}_G^\text{fin}(T) \|_G = \Omega_{\Pi,G}^\text{fin}(D) \).
2. There exists a bijection \( g : \| \mathcal{F}_T \|_G \rightarrow \mathcal{F}_{\Pi,G}^D \) such that, for every \( E \in \| \mathcal{F}_T \|_G \), \( \| \text{paths}_G^\text{fin}(T) \|_G = g(E) \cap \Omega_{\Pi,G}^\text{fin}(D) \).
3. For every \( E \in \| \mathcal{F}_T \|_G \), \( \| P_T \|_G(E) = \Pi_{\Pi,G}(g(E)) \).

The above theorem essentially tells us that there is an effective way of obtaining the probability space \( \Pi_G(D) \) from the probability space \( \mathcal{F}_T \), no matter which chase tree \( T \) we consider.

### 6 STRATIFIED NEGATION

We have seen in Section 4 that for generative Datalog programs without negation, there is no need to go beyond the probabilistic semantics induced by the simple grounder, i.e., the semantics induced by the simple grounder is as good as the semantics induced by any other grounder (Theorem 4.12). In this section, we establish an analogous result for the important class of generative Datalog-\* programs with stratified negation. But let us first recall this class.

**Generative Datalog with Stratified Negation**

Consider a GDatalog-\* program \( \Pi \). The dependency graph of \( \Pi \), denoted \( \text{dg}(\Pi) \), is a multigraph \((V,E)\), where (i) \( V = \text{sch}(\Pi) \), (ii) for each \( \rho \in \Pi \) with \( P \) being the predicate of \( H(\rho) \), and for every predicate \( R \) occurring in \( B^-(\rho) \) (resp., \( B^+(\rho) \)), there is a positive edge (\( R, P \)) (resp., a negative edge (\( R, P \))), and (iii) there are no other edges in \( E \). We say that \( P \) depends on \( R \) (relative to \( \Pi \)) if there is a path in \( \text{dg}(\Pi) \) from \( R \) to \( P \). A cycle in \( \text{dg}(\Pi) \) is a path from a predicate to itself. We say that the program \( \Pi \) has stratified negation if \( \text{dg}(\Pi) \) does not contain a cycle that goes through a negative edge. Henceforth, we refer to such kind of programs as GDatalog-\*\textsuperscript{\*} programs.

Interestingly, when the negation is stratified, a certain stratification of the given program can be induced that guarantees that a negative literal is affected only by rules from previous strata. We proceed to formalize this. A strongly connected component of \( \text{dg}(\Pi) = (V,E) \) is a subset-maximal set \( C \subseteq V \) such that \( P \) depends on \( R \) for every two distinct predicates \( P, R \in C \). Let \( \text{sc}(\Pi) \) be the set of strongly connected components of \( \text{dg}(\Pi) \). A topological ordering of \( \text{sc}(\Pi) \) is a linear order \( C_1, \ldots, C_n \), for \( n = |\text{sc}(\Pi)| \), over \( \text{sc}(\Pi) \) such that, for \( i \leq j \leq n \), there is no predicate of \( C_j \) that depends on one of \( C_i \). We denote by \( \Pi|_{C_i} \) the subset of \( \Pi \) that keeps only the rules whose head mentions a predicate from \( C_i \).

**Perfect Grounding for GDatalog-\*\textsuperscript{\*} \( \{\Lambda\} \)**

Unlike positive programs, we can show that for generative Datalog-\*\textsuperscript{\*} programs there exists a better grounder than the simple one. More precisely, for a GDatalog-\*\textsuperscript{\*} program \( \Pi \), and a database \( D \) of edb(\( \Pi \)), there exists a grounder \( G \) of \( \Pi[D] \) such that \( \Pi_G(D) \) is as good as \( \Pi_{\text{SimpleG}}(D) \), but not vice versa. We proceed to introduce a grounder that leads to the ultimate semantics in the case of stratified negation.

Given a GDatalog-\*\textsuperscript{\*} \( \{\Lambda\} \) program \( \Pi \), we define the so-called perfect grounder \( \text{GPerfect}_\Pi \), which can be taken as the version of the simple grounder (see Definition 4.4) that considers the rules of \( \Pi \) in a certain order according to the induced stratification of \( \Pi \). Similarly to the simple grounder, \( \text{GPerfect}_\Pi \) relies on an operator that operates on (infinitary) existential-free TGD-\* programs. Consider such a program \( \Sigma \). We define \( \text{Perfect}_\Sigma \) as the monotonic function from \( 2^{\text{ground}(\Sigma)} \) to \( 2^{\text{ground}(\Sigma)} \) such that, for every \( \Sigma' \in 2^{\text{ground}(\Sigma)} \),

\[
\text{Perfect}_\Sigma(\Sigma') = \Sigma' \cup \{ h(\sigma) \mid \sigma \in \Sigma, \ h(B^-(\sigma)) \subseteq \text{heads}(\Sigma') \}
\]

The set \( \text{Perfect}_\Sigma(\Sigma') \) is defined as expected; due to the monotonicity of \( \text{Perfect}_\Sigma \), it is the least fixpoint of \( \text{Perfect}_\Sigma \) that contains \( \Sigma' \).

Assume now that \( C_1, \ldots, C_n \) is a topological ordering over \( \text{sc}(\Pi) \). For a set \( \Sigma \in 2^{\text{ground}(\Sigma)} \), we define \( \Sigma \uparrow C_i \in 2^{\text{ground}(\Sigma)} \), which, intuitively speaking, is the set assigned to \( \Sigma \) by the perfect grounder considering the rules stratum by stratum up to the \( i \)-th stratum. Eventually, \( \text{GPerfect}_\Pi \) will assign to \( \Sigma \) the (infinitary) TGD-\* program \( \Sigma \uparrow C_n \). By convention, let \( \Sigma \uparrow C_0 = \emptyset \). Then, for every \( i \in [n] \), with \( \Sigma_i = \Sigma_{\downarrow C_i} \cup \Sigma \uparrow C_{i-1} \cup \Sigma \),

\[
\Sigma \uparrow C_i = \begin{cases} \text{Perfect}_{\Sigma_{\downarrow C_i}}(\emptyset) \cup \Sigma, & \text{if } \text{AtR}_\Sigma \hookrightarrow \Sigma \uparrow C_{i-1} \\ \Sigma \uparrow C_{i-1}, & \text{otherwise}. \end{cases}
\]

We are now ready to introduce the function \( \text{GPerfect}_\Pi \).

**Definition 6.1 (Perfect Grounder).** Consider a GDatalog-\*\textsuperscript{\*} \( \{\Lambda\} \) program \( \Pi \), and let \( C_1, \ldots, C_n \) be a topological ordering over \( \text{sc}(\Pi) \).

The function \( \text{GPerfect}_\Pi : [2^{\text{ground}(\Sigma)}]_\Pi \rightarrow 2^{\text{ground}(\Sigma)} \) is such that \( \text{GPerfect}_\Pi(\Sigma) = \Sigma \uparrow C_n \), for every \( \Sigma \in [2^{\text{ground}(\Sigma)}]_\Pi \).

An example that illustrates \( \text{GPerfect}_\Pi \) can be found in the appendix. We can show that \( \text{GPerfect}_\Pi \) is indeed a grounder. Note that this relies on a lemma that establishes the following: every (infinitary) TGD-\* program \( \Sigma \) that belongs to the \( \text{GPerfect}_\Pi \)-grounder of \( \Pi \) has a unique stable model, namely the instance \( \text{heads}(\Sigma) \).

**Proposition 6.2.** Consider a GDatalog-\*\textsuperscript{\*} \( \{\Lambda\} \) program \( \Pi \). It holds that \( \Pi_{\text{GPerfect}_\Pi} \) is a grounder of \( \Pi \).

We now show the main result of this section: the perfect grounder leads to the ultimate semantics when the negation is stratified.

**Theorem 6.3.** Consider a GDatalog-\*\textsuperscript{\*} \( \{\Lambda\} \) program \( \Pi \) and a database \( D \) of edb(\( \Pi \)). Let \( G \) be an arbitrary grounder of \( \Pi[D] \). It holds that \( \Pi_{\text{GPerfect}_\Pi(D)} \) is as good as \( \Pi_G(D) \).

### 7 CONCLUSIONS

We have introduced generative Datalog with negation that allows sampling from discrete probability distributions and defined its probabilistic semantics by interpreting negation according to the standard stable model semantics. We have also shown that the semantics can be equivalently defined via a fixpoint procedure based on a new chase procedure that operates on ground programs. Here, we focused on the generative component, but one may also incorporate conditioning under events of positive probability just as in [3]. An interesting direction for future research, which is highly relevant for practical implementations, is to devise sophisticated grounders that avoid as much as possible the generation of superfluous ground rules. Furthermore, in view of the fact that continuous distributions appear in a variety of application scenarios for probabilistic databases, it is natural to extend generative Datalog with negation to support continuous distributions.
ACKNOWLEDGMENTS

We thank the anonymous referees for their useful feedback. Alviano was partially supported by the Italian Ministry of Research (MUR) under PNRR project FAIR “Future AI Research”, CUP H23C22000860006, by the LAIA lab (part of the SILA labs), and by GNCS-IndAM. Lanzinger was supported by the Royal Society “RAISON DATA” project (Reference No. RP/R1/201074). Pieris was supported by the EPSRC grant EP/S003800/1 "EQUID".

REFERENCES

[1] Mario Alviano, Michael Morak, and Andreas Pieris. 2017. Stable Model Semantics for Tuple-Generating Dependencies Revisited. In PODS 377–388.
[2] Chitta Baral, Michael Gelfond, and J. Nelson Rushin. 2009. Probabilistic reasoning with answer sets. Theory Pract. Log. Program. 9, 1 (2009), 57–144.
[3] Vince Bárán, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. ACM Trans. Database Syst. 42, 4 (2017), 22:1–22:35.
[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhun, Theofanis Karatatzos, Rohit Singh, Paul A. Sernip, Paul Horold, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. J. Mach. Learn. Res. 20 (2019), 28:1–28:6.
[5] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog++: A Family of Logical Knowledge Representation and Query Languages for New Applications. In LICS 228–242.
[6] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, and Noah D. Goodman. 2019. Stan: A Probabilistic Programming Language. Journal of Statistical Software 76, 1 (2017), 1–32.
[7] Fabio Gagliardi Cozman and Denis Deratani Mauá. 2020. The joy of Probabilistic Answer Set Programming: Semantics, complexity, expressivity, inference. Int. J. Approx. Reason. 125 (2020), 218–239.
[8] Daniel Deutch, Christoph Koch, and Tova Milo. 2010. On probabilistic fixpoint and Markov chain query languages. In PODS 215–226.
[9] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sh. Shtetov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Program. 15, 3 (2015), 358–401.
[10] Norbert Fuhr. 1995. Probabilistic Datalog - A Logic For Powerful Retrieval Methods. In SIGIR 282–290.
[11] Norbert Fuhr. 2000. Probabilistic datalog: Implementing logical information retrieval for advanced applications. J. Am. Soc. Inf. Sci. 51, 2 (2000), 95–110.
[12] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In ICLP/SLP 1070–1080.
[13] Michael Gelfond and Vladimir Lifschitz. 1991. Classical Negation in Logic Programs and Disjunctive Databases. New Gener. Comput. 9, 3/4 (1991), 365–386.
[14] Noah D. Goodman,ika, Michael Morak, and Andreas Pieris. 2017. Stable Model Semantics for Tuple-Generating Dependencies Revisited. In PODS 377–388.
[20] Thomas Lukasiewicz. 2001. Probabilistic logic programming with conditional constraints. ACM Trans. Comput. Log. 2, 3 (2001), 289–339.
[21] Brian Milch, Bhaskara Marthi, Stuart Russell, David A. Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In JICAI 1352–1359.
[22] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In AAAI 2476–2482.
[23] Arvi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. Technical Report, Charles River Analytics.
[24] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. 2016. Statis-
tical Relational Artificial Intelligence: Logic, Probability, and Computation. Morgan & Claypool Publishers.
[25] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilis-
tic Prolog and Its Application in Link Discovery. In IJCAI 2462–2467.
[26] Matthew Richardson and Pedro M. Domingos. 2006. Markov Logic Networks. Mach. Learn. 62, 1-2 (2006), 107–136.
[27] N. Saheb-Djahromi. 1978. Probabilistic LCF. In MPCS 442–451.
[28] Parag Singla and Pedro M. Domingos. 2007. Markov Logic in Infinite Domains. In UAI 368–375.
[29] David Toulpin, Jan-Willem van de Meent, and Frank D. Wood. 2015. Probabilistic Programming in Anglican. In ECML PKDD 308–311.
[30] Hui Wan and Michael Kifer. 2009. Belief Logic Programming: Incremental Computation with Correlation of Evidence. In LPNMR 316–328.
We consider a simple scenario in which a given set of dimes are tossed, and if none of them show tail then also a given set of quarters are tossed. Let \( \Delta = \{ \text{Flip} \} \) and assume we have the following schema:

\[
\begin{align*}
\text{Dime}(\text{dime}_id) & \quad \text{Quarter}(\text{quarter}_id) & \quad \text{DimeTail}(\text{dime}_id, \text{boolean}) & \quad \text{QuarterTail}(\text{quarter}_id, \text{boolean}) & \quad \text{SomeDimeTail}.
\end{align*}
\]

For simplicity, let us assume global identifiers, that is, the sets of identifiers of dimes and quarters have empty intersection. The GDatalog\(^{\text{st}}\)[\(\Delta\)] program \(\Pi\) over the above schema that encodes our simple scenario is as follows:

\[
\begin{align*}
\text{Dime}(x) & \rightarrow \text{DimeTail}(x, \text{Flip}(0.5)[x]) \\
\text{DimeTail}(x, 1) & \rightarrow \text{SomeDimeTail} \\
\text{Quarter}(x), \neg\text{SomeDimeTail} & \rightarrow \text{QuarterTail}(x, \text{Flip}(0.5)[x]).
\end{align*}
\]

Hence, the TGD\(^{\text{st}}\) program \(\Sigma_{\Pi}\) is the following:

\[
\begin{align*}
\text{Dime}(x) & \rightarrow \text{Active}_1^{\text{Flip}}(0.5, x) \\
\text{DimeTail}(x, 1) & \rightarrow \text{SomeDimeTail} \\
\text{Quarter}(x), \neg\text{SomeDimeTail} & \rightarrow \text{Active}_1^{\text{Flip}}(0.5, x) \\
\text{Active}_1^{\text{Flip}}(0.5, x) & \rightarrow \exists z \text{Result}_1^{\text{Flip}}(0.5, x, z) \\
\text{Result}_1^{\text{Flip}}(0.5, x, z), \text{Dime}(x) & \rightarrow \text{DimeTail}(x, z) \\
\text{Result}_1^{\text{Flip}}(0.5, x, z), \text{Quarter}(x), \neg\text{SomeDimeTail} & \rightarrow \text{QuarterTail}(x, z).
\end{align*}
\]

The dependency graph \(\text{dg}(\Pi)\) is shown in Figure 1. In the following, let us consider the topological ordering

\[
C_1 = \{ \text{Dime} \} \quad C_2 = \{ \text{Quarter} \} \quad C_3 = \{ \text{DimeTail} \} \quad C_4 = \{ \text{SomeDimeTail} \} \quad C_5 = \{ \text{QuarterTail} \}
\]

and the database

\[
D = \{ \text{Dime}(1), \text{Dime}(2), \text{Quarter}(3) \}.
\]

Note that \(\Pi_{|C_1} \cup \Pi_{|C_2} = \{ \rightarrow \alpha \ | \ \alpha \in D \}, \Pi_{|C_3} = \{ \text{Dime}(x) \rightarrow \text{DimeTail}(x, \text{Flip}(0.5)[x]) \}, \Pi_{|C_4} = \{ \text{DimeTail}(x, 1) \rightarrow \text{SomeDimeTail} \},\) and \(\Pi_{|C_5} = \{ \text{Quarter}(x), \neg\text{SomeDimeTail} \rightarrow \text{QuarterTail}(x, \text{Flip}(0.5)[x]) \}.\) Hence, the perfect grounding for \(\Pi[D]\) works with \(D\) and

\[
\begin{align*}
\Sigma_{\Pi_{|C_3}} = \left\{ \begin{array}{c}
\text{Dime}(x) \rightarrow \text{Active}_1^{\text{Flip}}(0.5, x) \\
\text{Active}_1^{\text{Flip}}(0.5, x) \rightarrow \exists z \text{Result}_1^{\text{Flip}}(0.5, x, z) \\
\text{Result}_1^{\text{Flip}}(0.5, x, z), \text{Dime}(x) \rightarrow \text{DimeTail}(x, z)
\end{array} \right\} \\
\Sigma_{\Pi_{|C_4}} = \left\{ \begin{array}{c}
\text{DimeTail}(x, 1) \rightarrow \text{SomeDimeTail}
\end{array} \right\} \\
\Sigma_{\Pi_{|C_5}} = \left\{ \begin{array}{c}
\text{Quarter}(x), \neg\text{SomeDimeTail} \rightarrow \text{Active}_1^{\text{Flip}}(0.5, x) \\
\text{Active}_1^{\text{Flip}}(0.5, x) \rightarrow \exists z \text{Result}_1^{\text{Flip}}(0.5, x, z) \\
\text{Result}_1^{\text{Flip}}(0.5, x, z), \text{Quarter}(x), \neg\text{SomeDimeTail} \rightarrow \text{QuarterTail}(x, z)
\end{array} \right\}.
\end{align*}
\]

We first consider the case where the first dime shows tail and the second shows head, which is encoded by the following \(\Sigma \in [2^{\text{ground}(\Sigma_{\Pi})}]_\alpha:\)

\[
\begin{align*}
\text{Active}_1^{\text{Flip}}(0.5, 1) & \rightarrow \text{Result}_1^{\text{Flip}}(0.5, 1, 1) \\
\text{Active}_1^{\text{Flip}}(0.5, 2) & \rightarrow \text{Result}_1^{\text{Flip}}(0.5, 2, 0).
\end{align*}
\]
The perfect grounding is obtained as follows:

\[ \Sigma \uparrow C_2 = \begin{cases} 
\rightarrow & \text{Dime}(1) \\
\rightarrow & \text{Dime}(2) \\
\rightarrow & \text{Quarter}(3) 
\end{cases} \]

\[ \Sigma \uparrow C_3 = \Sigma \uparrow C_2 \cup \left\{ \begin{array}{l}
\text{Dime}(1) \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 1) \\
\text{Dime}(2) \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 2) \\
\text{Result}_1^{\text{Flip}}(0.5, 1, 1), \text{Dime}(1) \rightarrow \text{DimeTail}(1, 1) \\
\text{Dime}(2) \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 2) \\
\text{Result}_1^{\text{Flip}}(0.5, 2, 0), \text{Dime}(2) \rightarrow \text{DimeTail}(2, 0) 
\end{array} \right. \]

\[ \Sigma \uparrow C_4 = \Sigma \uparrow C_3 \cup \{ \text{DimeTail}(1, 1) \rightarrow \text{SomeDimeTail} \} \]

\[ \Sigma \uparrow C_5 = \Sigma \uparrow C_4 \cup \emptyset = \text{GPerfect}_{\Pi[D]}(\Sigma). \]

Note that \( \text{AtR}_\Sigma \hookrightarrow \text{GPerfect}_{\Pi[D]}(\Sigma)\), i.e., \( \Sigma \in \text{terminals}(\text{GPerfect}_{\Pi[D]}(\Sigma)) \). Moreover, \( \Sigma \cup \text{GPerfect}_{\Pi[D]}(\Sigma) \) belongs to \( \text{GPerfect}_{\Pi[D]}\)-ground(\( \Pi[D] \)), and \( \text{sms}(\Sigma \cup \text{GPerfect}_{\Pi[D]}(\Sigma) \cup \text{GPerfect}_{\Pi[D]}(\Sigma)) = \{ \text{Dime}(1), \text{Dime}(2), \text{Quarter}(3), \text{Active}^{\text{Flip}}_1(0.5, 1), \text{DimeTail}(1, 1), \text{Active}^{\text{Flip}}_1(0.5, 2), \text{DimeTail}(2, 0), \text{SomeDimeTail} \} = \{ \text{heads}(\Sigma \cup \text{GPerfect}_{\Pi[D]}(\Sigma)) \}. \]

We now consider the case in which no dime shows tail, which is encoded by the following \( \Sigma \in \text{ground}(\Sigma^2) \):

\[ \text{Active}^{\text{Flip}}_1(0.5, 1) \rightarrow \text{Result}^{\text{Flip}}_1(0.5, 1, 0) \]

\[ \text{Active}^{\text{Flip}}_1(0.5, 2) \rightarrow \text{Result}^{\text{Flip}}_1(0.5, 2, 0). \]

The perfect grounding is obtained as follows:

\[ \Sigma \uparrow C_2 = \begin{cases} 
\rightarrow & \text{Dime}(1) \\
\rightarrow & \text{Dime}(2) \\
\rightarrow & \text{Quarter}(3) 
\end{cases} \]

\[ \Sigma \uparrow C_3 = \Sigma \uparrow C_2 \cup \left\{ \begin{array}{l}
\text{Dime}(1) \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 1) \\
\text{Dime}(2) \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 2) \\
\text{Result}_1^{\text{Flip}}(0.5, 1, 0), \text{Dime}(1) \rightarrow \text{DimeTail}(1, 0) \\
\text{Result}_1^{\text{Flip}}(0.5, 2, 0), \text{Dime}(2) \rightarrow \text{DimeTail}(2, 0) 
\end{array} \right. \]

\[ \Sigma \uparrow C_4 = \Sigma \uparrow C_3 \cup \emptyset \]

\[ \Sigma \uparrow C_5 = \Sigma \uparrow C_4 \cup \{ \text{Quarter}(3), \neg \text{SomeDimeTail} \rightarrow \text{Active}^{\text{Flip}}_1(0.5, 3) \} = \text{GPerfect}_{\Pi[D]}(\Sigma). \]

Note that \( \text{AtR}_\Sigma \hookrightarrow \text{GPerfect}_{\Pi[D]}(\Sigma) \) does not hold due to \( \text{Active}^{\text{Flip}}_1(0.5, 3) \in \text{heads}(\Sigma \cup \text{GPerfect}_{\Pi[D]}(\Sigma)) \).