From Probabilistic NetKAT to ProbLog: New Algorithms for Inference and Learning in Probabilistic Networks

BIRTHE VAN DEN BERG*, KU Leuven, Belgium
TIMOTHY VAN BREMEN*, KU Leuven, Belgium
VINCENT DERKINDEREN*, KU Leuven, Belgium
ANGELIKA KIMMIG, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium
LUC DE RAEDT, KU Leuven, Belgium

We consider the problem of modelling *probabilistic networks*, where one is interested in modelling packets moving through an unreliable network, with each link carrying a certain probability of failure. The domain-specific language Probabilistic NetKAT provides a clear syntax and semantics for specifying such networks in a filter-based (rather than state-based) manner. We introduce a formal translation to transform this domain-specific language into ProbLog, a popular probabilistic logic programming language. We show how employing this translation allows one to take advantage of different inference and learning procedures designed for ProbLog in the context of Probabilistic NetKAT. This illustrates how transforming domain-specific languages to general-purpose probabilistic programming languages can provide a kind of rapid prototyping.

1 INTRODUCTION

In network management, *software-defined networking* provides a paradigm for defining network configurations by specifying the characteristics of a network and its routing policy programmatically. Recently, Anderson et al. [2014] proposed *NetKAT*, a network modelling language with a formal semantics for this. *Probabilistic NetKAT* [Foster et al. 2016] extends this language to allow for modelling networks in the presence of *uncertainty*: for example, modelling an unreliable network in which a packet may be dropped with a certain probability.

At the same time, the paradigm of *probabilistic logic programming* has enjoyed a rich history, with a simple and flexible semantics that is naturally suited to modelling networks. In this paper we explore the connection between the two paradigms by translating Probabilistic NetKAT programs to the probabilistic logic programming language ProbLog [Fierens et al. 2015], in such a way that the semantics of the original program are retained. This allows us to exploit several innovations from the world of probabilistic logic programming in Probabilistic NetKAT programs, including parameter learning, the use of semirings for modelling latency, and support for continuous distributions. We argue that in general, it is useful to prototype domain specific probabilistic languages by translating them to general-purpose probabilistic languages and thus benefiting from the range of inference tasks they provide.

2 RELATED WORK

Gehr et al. [2018] present a two-part approach, *Bayonet*, comprising a probabilistic programming language, along with a corresponding inference system, that is geared towards modelling probabilistic networks like those in this paper. In a similar spirit to our approach, they use a translation

*equal contribution

Authors' addresses: Birthe van den Berg, KU Leuven, Belgium, birthe.vandenberg@cs.kuleuven.be; Timothy van Bremen, KU Leuven, Belgium, timothy.vanbremen@cs.kuleuven.be; Vincent Derkinderen, KU Leuven, Belgium, vincent.derkinderen@cs.kuleuven.be; Angelika Kimmig, KU Leuven, Belgium, angelika.kimmig@cs.kuleuven.be; Tom Schrijvers, KU Leuven, Belgium, tom.schrijvers@cs.kuleuven.be; Luc De Raedt, KU Leuven, Belgium, luc.deraedt@cs.kuleuven.be.

Actions		
$a ::=$	t	Test
$\mid$	$x \leftarrow n$	Modification
$\mid$	$a_1 \mathbin{\&} a_2$	Parallel
$\mid$	$a_1 \,;\, a_2$	Sequence
$\mid$	$a_1 \oplus_r a_2$	Choice
$\mid$	$a^\star$	Iteration
$\mid$	dup	Duplication

Tests		
$t ::=$	$t_1 \mathbin{\&} t_2$	Disjunction
$\mid$	$t_1 \,;\, t_2$	Conjunction
$\mid$	$\overline{t}$	Negation
$\mid$	skip	True
$\mid$	drop	False
$\mid$	$x = n$	Filter

Fig. 2. ProbNetKAT syntax.

system but target the language PSI [Gehr et al. 2016] rather than ProbLog. Bayonet is different from Probabilistic NetKAT, as it reasons over states rather than packet histories, with the latter more easily capturing temporal properties [Gehr et al. 2018]. The usage of ProbLog enables learning programs from data and extensions such as using semirings for a variety of other tasks. In the specific context of Probabilistic NetKAT, Smolka et al. [2017] developed an alternative characterization of the semantics of Probabilistic NetKAT and also proposed an interpreter for the language in OCaml.

3 BACKGROUND

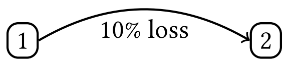
In this section, we review some background on Probabilistic NetKAT, as well as the probabilistic logic programming language ProbLog.

3.1 Probabilistic NetKAT

Probabilistic NetKAT [Foster et al. 2016] extends NetKAT [Anderson et al. 2014], a high-level network modelling language—which in turn is based on Kleene algebra with tests (KAT)—with probabilistic behavior.

Syntax. The syntax of Probabilistic NetKAT (ProbNetKAT) is shown in Figure 2. A NetKAT program is an action a, which can take one of several forms: a modification $x \leftarrow n$ that assigns value n to field x of the current packet; parallel composition $a_1 \mathbin{\&} a_2$; sequential composition $a_1 \,;\, a_2$; probabilistic choice $a_1 \oplus_r a_2$ that chooses a_1 with probability r and a_2 with probability $(1 - r)$; iteration $a^\star$; duplication *dup* of the current packet; or a test t. Tests take into account the packet history, which tracks packets π during their travel from node to node. A test t can be a disjunction $t_1 \mathbin{\&} t_2$; a conjunction $t_1 \,;\, t_2$; a negation $\overline{t}$; *skip*, which is the set H of all packet histories and always true; *drop*, which is the empty set and always false; or $x = n$, which filters the packet history to only include packets π where field x has value n.

The ProbNetKAT example in Figure 1 encodes the forwarding of packets in a network from switch 1 to switch 2 with a 10% loss factor. More examples of ProbNetKAT programs are given in the appendix to this paper.



```
(sw = 1 ; sw <- 2 ⊕0.9 drop)
```

Fig. 1. Forwarding packets with 10% loss.

Semantics. Due to the combination of unbounded iteration with probabilistic choice, ProbNetKAT's semantics features continuous distributions, which are modeled with Borel sets. Yet, for the sake of the translation to ProbLog we bound the iteration and restrict ourselves to discrete distributions; this yields a simplified semantics for actions.

$$[\![a]\!] = H \to \mathcal{D}\, H$$

Here H is the set of *all* packet histories. A packet is a collection of fields x with values n, which are natural numbers. A packet history $\eta \in H$ is a non-empty sequence $\langle \pi_1, \ldots, \pi_n \rangle$ of packets, from youngest to oldest, that tracks the subsequent changes to a packet. With $\mathcal{D}\,H$ we denote the set of discrete distributions over H and we write $[\![a]\!](\eta_{in}, \eta_{out})$ to denote the probability that, when a is run on η_{in}, the result is η_{out}.

3.2 ProbLog

ProbLog is a probabilistic logic programming language that combines the classic logic programming language Prolog with uncertainty, by allowing facts to be annotated with independent probabilities. The resulting semantics follows Sato's *distribution semantics* [Sato 1995]. Formally, a ProbLog program can be viewed as a triple $(\mathcal{F}, p, \Psi)$ where $\mathcal{F}$ is a finite set of ground atoms called *probabilistic facts*, $p : \mathcal{F} \longrightarrow [0, 1]$ is a labelling function assigning a probability to each probabilistic fact, and Ψ is a *stratified normal logic program*. The probability of a goal $P_{\mathcal{F},p,\Psi}(G)$ for a ground goal atom G can be defined as:

$$P_{\mathcal{F},p,\Psi}(G) = \sum_{\mathcal{F}' \subseteq \mathcal{F}, G \in \Pi(\mathcal{F}')} \prod_{f \in \mathcal{F}'} p(f) \prod_{f \in \mathcal{F} \setminus \mathcal{F}'} (1 - p(f)) \qquad (1)$$

where $\Pi(\mathcal{F}')$ denotes the unique stable model of $\Pi \cup \mathcal{F}'$. When clear from context, we will write $P_{\mathcal{F},p,\Pi}(G)$ as $P(G)$. For the full details of the ProbLog syntax and semantics, including semantics for computing the probability of non-ground goal atoms, we refer the reader to Fierens et al. [2015].

Consider the example program in Figure 3 where `alarm` is true iff `alarm_on` and there is either a `burglary` or an `earthquake`. The probabilistic facts $\mathcal{F}$ with their respective probabilities p are indicated in the first three lines. The deterministic part of the program is given in the last two lines. The three probabilistic facts result in $2^3 = 8$ possible worlds. Only three of these worlds entail the truth of `alarm`. Thus, to know the probability of `alarm` we compute $P(\texttt{alarm}) = 0.3(0.5)(0.4) + 0.3(1 - 0.5)(0.4) + 0.3(0.5)(1 - 0.4) = 0.21$.

```
0.4::burglary.
0.5::earthquake.
0.3::alarm_on.
alarm :- alarm_on, burglary.
alarm :- alarm_on, earthquake.
```

Fig. 3. ProbLog example

4 TRANSLATION

Figure 4 defines a family of translation functions $\lfloor \cdot \rfloor$ to map the various ProbNetKAT syntax constructs to their corresponding ProbLog encodings.

Packets and Packet Histories. A packet π is translated to a ProbLog list of label-value pairs. A packet history η is similarly translated to a ProbLog list.

Tests. Tests correspond naturally to ProbLog goals that possibly take a packet history as an argument. The translation function takes an extra parameter H which is a ProbLog variable that acts as a placeholder for this packet history. All but one test of the test constructs have primitive ProbLog counterparts.[1] Only the membership test requires a ProbLog helper predicate `mem/3`; its definition is given in Appendix A.1.

Actions. Finally, ProbNetKAT actions a are translated to a pair $\langle G, R\rangle$ of a ProbLog goal G together with a set of additional ProbLog rules R that define auxiliary predicates used by the goal.

The ProbNetKat semantics of actions is to probabilistically transform a packet history. The probabilistic aspect of this is ingrained in ProbLog's semantics, but the packet history transformation needs to be encoded explicitly. For that purpose the generated ProbLog goal G takes an incoming

[1]Note the difference in the meaning of the semicolon in ProbLog queries and ProbNetKAT tests: the former is disjunction and the latter conjunction.

$$\boxed{\textbf{Packets and Packet Histories}}$$

$$\lfloor\{x_1 = n_1, \ldots, x_k = n_k\}\rfloor = \texttt{[}x_1 \texttt{-} n_1, \ldots, x_k \texttt{-} n_k\texttt{]} \qquad \lfloor\langle\pi_1, \ldots, \pi_n\rangle\rfloor = \texttt{[}\,\lfloor\pi_1\rfloor, \ldots, \lfloor\pi_1\rfloor\texttt{]}$$

$$\boxed{\textbf{Tests}}$$

$$\lfloor skip\rfloor_H = \texttt{true} \qquad \lfloor\{x = n\}\rfloor_H = \texttt{mem(x,n,H)} \qquad \lfloor t_1 \,\&\, t_2\rfloor_H = \lfloor t_1\rfloor_H; \lfloor t_2\rfloor_H$$

$$\lfloor drop\rfloor_H = \texttt{false} \qquad \lfloor\bar{t}\rfloor_H = \texttt{not(}\lfloor t\rfloor_H\texttt{)} \qquad \lfloor t_1 \,;\, t_2\rfloor_H = \lfloor t_1\rfloor_H, \lfloor t_2\rfloor_H$$

$$\boxed{\textbf{Actions}}$$

$$\lfloor t\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(\lfloor t\rfloor_{H_{In}}, \texttt{HIn = HOut, CIn = COut}), \emptyset\rangle$$

$$\lfloor x \leftarrow n\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(\texttt{modifyH(x, n, HIn, HOut), CIn = COut}), \emptyset\rangle$$

$$\lfloor dup\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(\texttt{duplicate(HIn, HOut), CIn = COut}), \emptyset\rangle$$

$$\lfloor a_1 \,\&\, a_2\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(G_1 \,;\, G_2),\, R_1 \cup R_2\rangle \text{with } \lfloor a_1\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle G_1, R_1\rangle \text{ and } \lfloor a_2\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle G_2, R_2\rangle$$

$$\lfloor a_1 \,;\, a_2\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(G_1 \,,\, G_2),\, R_1 \cup R_2\rangle \text{with } \lfloor a_1\rfloor^{H_{Mid},C_{Mid}}_{H_{In},C_{In}} = \langle G_1, R_1\rangle \text{ and } \lfloor a_2\rfloor^{H_{Out},C_{Out}}_{H_{Mid},C_{Mid}} = \langle G_2, R_2\rangle$$

$$\lfloor a_1 \oplus_r a_2\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle(\texttt{f(CIn), G1; not(f(CIn)), G2}), R_1 \cup R_2 \cup R_3\rangle$$

$$\text{with } \lfloor a_1\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle G_1, R_1\rangle \text{ and } \lfloor a_2\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle G_2, R_2\rangle$$

$$R_3 = \texttt{r::f(CIn).} \text{ with } \texttt{f} \text{ fresh}$$

$$\lfloor a^\star\rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}} = \langle\texttt{f(HIn, CIn, HOut, COut)}, R \cup R_1 \cup R_2\rangle$$

$$\text{with } \lfloor a\rfloor^{H_2,C_2}_{H_1,C} = \langle G, R\rangle \text{ and } H, H_1, H_2, H_3, C, C_1, C_2, C_3 \text{ free and } \texttt{f/4} \text{ fresh}$$

$$R_1 = \texttt{f(H,C,H,C).}$$

$$R_2 = \texttt{f(H1,C1,H3,C3) :- C is C1-1, G, f(H2,C2,H3,C3).}$$

Fig. 4. Formal translation of ProbNetKat to ProbLog

and an outgoing packet history (`HIn` and `HOut`, respectively) as explicit arguments. At the meta-level, the translation function is parameterized in the names of these arguments too.

As tests do not modify the packet history, their output history is the same as their input history. Modification and duplication are handled by two auxiliary ProbLog predicates given in Appendix A.1. Parallel and sequential composition are mapped to ProbLog disjunction and conjunction; both thread the packet history appropriately. Probabilistic choice is mapped to a ProbLog disjunction whose branches are guarded by a probabilistic fact with the corresponding probability, and Kleene iteration is mapped to a recursive ProbLog rule.

There is one further complication in the mapping of probabilistic choices inside Kleene iterations. According to the ProbNetKAT semantics, the dynamic repetitions of a probabilistic choice are all independent. In contrast, repeated observations of the same probabilistic ProbLog fact are either consistently true or consistently false. Hence, making independent choices in ProbLog requires distinct facts. We accomplish this with parameterized facts, to which we supply a different parameter at different dynamic invocations. For simplicity, we parameterize all probabilistic facts (not only those inside Kleene iterations) in a counter value that we thread through the execution (`CIn` and `COut`). The counter is decremented at the start of every iteration. Hence, repeated observations of probabilistic facts get a different counter value and thus are independent.

All translated ProbNetKAT programs consist of helper predicates (Appendix A.1) along with the program-specific generated code. Recall the ProbNetKAT action from Figure 1. This translates to the following ProbLog program $\lfloor a \rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}}$ (improved for readability; the original is in Appendix A.2):

```
0.9::f(V1).
main(HIn,HOut,CIn) :- mem(sw,1,HIn),
                      (f(CIn), modifyH(sw,2,HIn,HOut) ; not(f(CIn)), false).
```

The main predicate has three parameters `HIn`, `HOut`, `CIn`, which are the same as those of the translation function. The `COut` parameter is not present as it is only needed for intermediate results, not for `main`. More examples are provided in Appendix A.3.

5 QUERIES

The ProbLog program resulting from the automated translation in the previous section can be used to query $[\![a]\!](\eta_{in}, \eta_{out})$ (recall the semantics from Section 3.1). This is best illustrated by an example. Consider the ProbNetKAT action and its ProbLog translation in the previous section. To obtain the probabilities of the possible output histories C given an input packet history $\langle\{sw = 1\}\rangle$, we can simply query `main([[sw-1]],HOut,1)`, yielding a probability of 0.9 for output $\langle\{sw = 2\}\rangle$. Note that the choice of `CIn` $= 1$ in this query is arbitrary: `CIn` simply denotes the counter starting value. We can also restrict the output histories to ones satisfying some property of interest, by defining and querying a predicate that holds whenever the property in question is true.

```
property_holds(HIn, CIn) :- main(HIn, HOut, CIn), property(HOut).
```

A key limitation of ProbLog's current inference mechanics is that the query and program must have a finite grounding [Fierens et al. 2015], stemming from the finite support condition of Sato's distribution semantics [Sato 1995]. The translation of a ProbNetKAT action containing a Kleene star will violate this condition. To get around this, we adopt the approximation scheme proposed by Foster et al. [2016], and alluded to in the simplified semantics of Section 3.1: we bound the number of Kleene star iterations by a finite number. In our automatic translation, this only requires bounding `C1` > 0 in the definition of $\lfloor a^\star \rfloor^{H_{Out},C_{Out}}_{H_{In},C_{In}}$. The number of Kleene star iterations is then determined by the counter starting value, `CIn`.

5.1 Expected Utility

The core ProbLog language has seen several different extensions over the years. One such extension is to allow for the computation of *expected utilities* [Derkinderen and De Raedt 2020; Eisner 2002]. In this setting, certain probabilistic facts—and, distinct from the original ProbLog setting, certain derived atoms as well—are annotated with a *utility*. In this context, we are typically no longer interested in the probability of a specific query, but instead wish to compute the expected utility of the program as a whole. This setting can be formalised as follows: instead of having only a mapping $p : \mathcal{F} \mapsto [0, 1]$, we now also associate probabilistic facts and certain derived atoms with utilities $u : C \mapsto \mathbb{R}$ for some finite subset $C \subseteq atoms(\Pi \cup \mathcal{F})$, where $atoms(\Pi \cup \mathcal{F})$ denotes the Herbrand base of $\Pi \cup \mathcal{F}$. The utility $u(\mathcal{F}')$ of a choice on the probabilistic facts $\mathcal{F}' \subseteq \mathcal{F}$ is the sum of the utilities of its derived atoms. The expected utility of the whole program $EU_{\mathcal{F},p,u,\Pi}$ is then defined as:

$$EU_{\mathcal{F},p,u,\Pi} = \sum_{\mathcal{F}' \subseteq \mathcal{F}} \overbrace{\prod_{f \in \mathcal{F}'} p(f) \prod_{f \in \mathcal{F} \setminus \mathcal{F}'} (1 - p(f))}^{p(\mathcal{F}')} \times \overbrace{\sum_{a \in C \cap \Pi(\mathcal{F}')} u(a)}^{u(\mathcal{F}')} \tag{2}$$

The expected utility can be used to model, for example, expected latency or expected cost. The brief example below illustrates how to compute the expected latency of packet $\langle\{sw = 1\}\rangle$ moving through a network. In this example there are two routes leading to $\langle\{sw = 2\}\rangle$, each with a different latency (five and ten seconds, respectively). In order to associate a specific route with a latency (utility), the route taken by the packet must be extractable from the packet history `HOut`. In the example below, this information is stored as packet attributes, `route1` and `route2`. Appendix A.4 includes an example where this information is instead extracted from the packet history.

$$(sw = 1 \ ; \ sw \leftarrow 2 \ ; \ route1 \leftarrow true \oplus_{0.9} route2 \leftarrow true)$$

```
0.9::f2(V1).
main(HIn,HOut,CIn) :- ...
visit_route1 :- input(In), main(In,HOut,1), mem(route1,true,HOut).
visit_route2 :- input(In), main(In,HOut,1), mem(route2,true,HOut).
utility(visit_route1, 5). % probability = 0.9
utility(visit_route2, 10). % probability = 0.1
input([[sw-1]]). % expected utility = 0.9 * 5 + 0.1 * 10 = 5.5
```

The mechanics of computing the expected utility is explored in more detail by Derkinderen and De Raedt [2020]. This approach uses *semirings* that overload ProbLog's sum and times operations. The same concept can also be used to solve a variety of other tasks, for example, computation of the "shortest path" in a network [Kimmig et al. 2017, 2011].

5.2 Learning

When Foster et al. [2016] introduced ProbNetKAT, learning programs from data was identified as an interesting direction for future research. They envisioned this as learning a network policy by observing (partial) traces of a running system. This is a type of parameter and structure learning problem that has been studied more generally in the context of probabilistic logic programming [De Raedt et al. 2015; Jain et al. 2019]. Transforming a ProbNetKAT action to a general probabilistic logic programming language can thus allow us to make use of new algorithms, enabling a variety of learning tasks. In ProbLog, we may address the task of *maximum-likelihood learning* of a program's probability parameters from (partially) observed models [Gutmann et al. 2011]. More formally, we are given a ProbLog program $(\mathcal{F}, \hat{p}, \Pi)$ in which the probability of (a subset of) the facts in $\mathcal{F}$ are unknown along with a set of models, and the goal is to complete $\hat{p}$.

In the context of ProbNetKAT, the learning of link reliability can be modelled as maximum-likelihood learning, and comes "for free" when translating to a ProbLog program. Consider the first ProbNetKAT example in Appendix A.3 where the 10% loss parameters might be unknown and could be learned from data. The data set of partially observed models comprises pairs of input and output packet histories, as illustrated by the two partially observed models below. Note that in the second model, the truth value of `main([[sw-1]],[[sw-2]],1)` is unobserved.

Model 1	Model 2
`main([[sw-2]],[[sw-1]],1).`	`not(main([[sw-2]],[[sw-1]],1)).`
`not(main([[sw-1]],[[sw-2]],1)).`	

6 CONCLUSION

We studied the network programming language ProbNetKAT, and showed how to transform programs in this language to the general-purpose probabilistic logic programming language ProbLog. We also examined some of the upsides of employing such an approach with respect to inference and learning. In particular, leveraging ProbLog's structure and parameter learning features can allow learning of networks, and making use of ProbLog's semiring extensions allows us to compute several interesting properties of networks modelled in ProbNetKAT.

ACKNOWLEDGMENTS

This work has received support from the Research Foundation – Flanders (1SA5520N and G095917N), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 694980), and the Flemish Government (AI Research Program).

REFERENCES

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *POPL*. ACM, 113–126.

Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. 2015. Inducing Probabilistic Relational Rules from Probabilistic Examples. In *IJCAI*. AAAI Press, 1835–1843.

Vincent Derkinderen and Luc De Raedt. 2020. Algebraic Circuits for Decision Theoretic Inference and Learning. In *ECAI (Frontiers in Artificial Intelligence and Applications)*, Vol. 325. IOS Press, 2569–2576.

Jason Eisner. 2002. Parameter Estimation for Probabilistic Finite-State Transducers. In *ACL*. ACL, 1–8.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, 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.

Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 282–309.

Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin T. Vechev. 2018. Bayonet: probabilistic inference for networks. In *PLDI*. ACM, 586–602.

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 62–83.

Bernd Gutmann, Ingo Thon, and Luc De Raedt. 2011. Learning the Parameters of Probabilistic Logic Programs from Interpretations. In *ECML/PKDD (1) (Lecture Notes in Computer Science)*, Vol. 6911. Springer, 581–596.

Arcchit Jain, Tal Friedman, Ondrej Kuzelka, Guy Van den Broeck, and Luc De Raedt. 2019. Scalable Rule Learning in Probabilistic Knowledge Bases. In *AKBC*.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. 2017. Algebraic model counting. *J. Appl. Log.* 22 (2017), 46–62.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. 2011. An Algebraic Prolog for Reasoning about Possible Worlds. In *AAAI*. AAAI Press, 209–214.

Taisuke Sato. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *ICLP*. MIT Press, 715–729.

Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor meets scott: semantic foundations for probabilistic networks. In *POPL*. ACM, 557–571.

A APPENDICES

A.1 Helper Predicates

```
mem(X,N,[P|_]) :- member(X, N, P).
member(X,N,[X-N|_]).
member(X,N,[_|H]) :- member(X,N,H).

modifyH(X,N,[P|H],[P1|H]) :- modify(X, N, P, P1).
modify(X, N, [], []).
modify(X, N, [X-_|H], [X-N|H]).
modify(X, N, [P-V|H], [P-V|H1]) :- P \== X, modify(X, N, H, H1).

duplicate([P | H], [P, P | H]).
```

A.2 Original Translation

```
0.9::f(V1).
main(HIn,HOut,CIn) :- mem(sw,1,HIn), V0 = HIn, V1 = CIn,
```

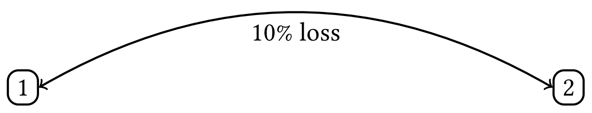


Fig. 5. Network with two nodes and a connection with 10% packet loss.

```
(f(V1), modifyH(sw,2,V0,HOut), COut = V1 ;
    not(f(V1)), false, HOut = V0, COut = V1).
```

A.3 Translation Examples

Here we show some more example translations.

First, Figure 5 pictures two interacting nodes that are connected via a network that has a 10% loss factor. We model this networking model in ProbNetKAT as follows:

```
(sw = 1; sw <- 2 ⊕_{0.9} drop) &
(sw = 2; sw <- 1 ⊕_{0.9} drop)
```

All translated ProbNetKAT programs consist of the helper predicates (Appendix A.1) and program-specific generated code. Here, we focus on the latter. For the example in Figure 5, the generated code consists of fresh facts `f2` and `f5`, parameterized by a counter and accompanied by their probability. The main predicate has three parameters `HIn, HOut, CIn`, which are the same as those of the translation function, except for the `COut` parameter which is only used internally.

```
0.9::f2(V1).
0.9::f5(V4).

main(HIn,HOut,CIn) :-
  mem(sw,1,HIn), V0 = HIn, V1 = CIn,
  (
    f2(V1), modifyH(sw,2,V0,HOut), COut = V1
  ;
    not(f2(V1)), false, HOut = V0, COut = V1
  )
;
  mem(sw,2,HIn), V3 = HIn, V4 = CIn,
  (
    f5(V4), modifyH(sw,1,V3,HOut), COut = V4
  ;
    not(f5(V4)), false, HOut = V3, COut = V4
  ).
```

Second, Figure 6 pictures a network with three nodes that send packets from node 1 to node 2 and node 2 to node 3, both with a 10% loss factor. The following program checks how many of the sent packets arrive at node 3.

```
((sw = 1 ; sw <- 2 ⊕_{0.9} drop) & (sw = 2 ; sw <- 3 ⊕_{0.9} drop))* ;
sw = 3
```

```
0.9::f15(V14).
0.9::f18(V17).
p2(V3,V8,V4,V9) :- V4 = V3, V9 = V8.
p2(V5,V10,V7,V12) :-
  V10 > 0, V8 is V10 - 1,
  (
    mem(sw,1,V5), V13 = V5, V14 = V8,
    (
      f15(V14), modifyH(sw,2,V13,V6), V11 = V14
    ;
      not(f15(V14)), false, V6 = V13, V11 = V14
    )
  ;
    mem(sw,2,V5), V16 = V5, V17 = V8,
    (
      f18(V17), modifyH(sw,3,V16,V6), V11 = V17
    ;
      not(f18(V17)), false, V6 = V16, V11 = V17
    )
  ),
  p2(V6,V11,V7,V12).

main(HIn,HOut,CIn) :-
  p2(HIn,CIn,V0,V1), mem(sw,3,V0), HOut = V0, COut = V1.
```

A.4 Expected Utility Example

The ProbNetKAT action below translates to the ProbLog program in Figure 7. An example of a query in this program is shown in Table 1.

$$\texttt{(sw = 1 ; dup ; (sw <- 2 } \oplus_{0.9} \texttt{ (sw <- 3 ; dup ; sw <- 2)))}$$

Notice that the route taken from $\langle\{sw = 1\}\rangle$ to $\langle\{sw = 2\}\rangle$ can be extracted from the packet history. We can utilise this to determine whether a link in the network was used, and to associate its usage with an increased latency (Figure 7). The expected latency in this program for the input packet $\langle\{sw = 1\}\rangle$ is 9.2.

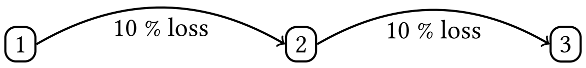


Fig. 6. Network with three nodes and connections with 10% packet loss.

```
% translated program
0.6::f4(V3).
main(HIn,HOut,CIn) :-
        mem(sw,1,HIn), (V0 = HIn), (V1 = CIn),
        duplicate(V0,V2), (V3 = V1),
        (
            f4(V3),modifyH(sw,2,V2,HOut),(COut = V3)
        ;
            not(f4(V3)),modifyH(sw,3,V2,V5),(V6 = V3),duplicate(V5,V7),
             (V8 = V6),modifyH(sw,2,V7,HOut),(COut = V8)
        ).
% extract link usage
used_link(X,Y) :- input(In), main(In, HOut, 1),
↪   contains_link(HOut,X,Y).
contains_link(H,X,Y) :- is_next_link(H,X,Y).
contains_link([H|P],X,Y) :- \+is_next_link(H,X,Y),
↪   contains_link(P,X,Y).
is_next_link([H,H2|_],X,Y) :- member(sw,X,H), member(sw,Y,H2).
is_next_link([H,H2|_],X,Y) :- member(sw,Y,H), member(sw,X,H2).
input([[sw-1]]).
% associate utilities (latency)
utility(used_link(1,2), 10).
utility(used_link(1,3), 4).
utility(used_link(3,2), 4).
```

Fig. 7. Example ProbLog program

Table 1. Results of querying `main([[sw-1]],HOut,1)` in the program of Figure 7.

Query	Probability
`main([[sw-1]],[[sw-2], [sw-1]],1)`	0.6
`main([[sw-1]],[[sw-2], [sw-3], [sw-1]],1)`	0.4