# From PRISM to ProbLog and Back Again

Alexander Vandenbroucke and Tom Schrijvers

KU Leuven
firstname.lastname@kuleuven.be

**Abstract.** PRISM and ProbLog are two prominent languages for Probabilistic Logic Programming. While they are superficially very similar, there are subtle differences between them that lead to different formulations of the same probabilistic model.
This paper aims to shed more light on the differences by developing two source-to-source transformations, from PRISM to ProbLog and back.

## 1 Introduction

Probabilistic Logic Programming (PLP) systems bring probabilistic modelling to the logic programming paradigm. Two well-known PLP systems are PRISM [8] and ProbLog [3].

At first glance, both systems are very similar. After all they have both been founded upon Sato's distribution semantics [7]. Moreover, they share the same Prolog syntax for programming with Horn clauses. However, appearances can be deceiving: both systems provide a subtly different approach for modelling in terms of the distribution semantics. While ProbLog features "named" probabilistic facts in a manner that is quite close to the distribution semantics, PRISM provides "anonymous" probabilistic facts in terms of distinct invocations of the built-in predicate `msw/2`. The latter is closer in approach to functional and imperative probabilistic languages and calculi [5, 10, 9].

This paper aims to shed more light on the subtle differences between ProbLog and PRISM. It does so by providing two source-to-source transformations, mapping PRISM programs to equivalent ProbLog programs and vice versa. Besides establishing that the two languages are equally expressive in terms of probabilistic modelling, the transformations reveal the essential differences between the two languages and the lengths one has to go to encode one in the other.

## 2 Background

In the introduction we mentioned that both ProbLog and PRISM implement Sato's distribution semantics [7], which itself subsumes the regular fixpoint semantics of logic programs [2]. This section briefly summarises how both systems implement this semantics.

*Semantics of Probabilistic Logic Programs* The semantics divides a program $P$ in two distinct parts: a set of probabilistic facts $\mathcal{F}$ and a set of rules $\mathcal{R}$ (logical clauses). A *total choice* $C$ is any subset of $\mathcal{F}$. A fact $f$ is said to be true in $C$ (false) if and only if $f \in C$ ($f \notin C$). A *total choice* $C$ and a set of clauses $\mathcal{R}$ together form a conventional logic program. We use $P \models a$ to denote that program $P$ logically entails an atom $a$ in the perfect model semantics [6].

Let $\mathcal{F} = \{p_1 :: f_1, \ldots, p_n :: f_n\}$, then the probability of a choice $C \subseteq \mathcal{F}$ is given by the product of the probabilities of the true and false facts:

$$P(C) = \prod_{p_i :: f_i \in C} p_i \times \prod_{p_i :: f_i \notin C} (1 - p_i)$$

The distribution semantics defines the probability of a query $q$ for a program $P = \mathcal{F} \cup \mathcal{R}$ as the sum of probabilities of all total choices that entail $q$:

$$P_P(q) = \sum_{C \subseteq \mathcal{F} \wedge C \cup \mathcal{R} \models q} P(C)$$

*ProbLog* ProbLog follows the above semantic fairly closely , denoting probabilistic facts and rules in the same way, e.g., as: (blue code denotes ProbLog code):

```
0.5 :: heads.
win :- heads.
```

However, ProbLog also provides advanced syntax, such as disjunctions annotated with probabilities (this syntax was inspired by the LPAD-language [4]):

```
0.5 :: heads ; 0.5 :: tails.
```

This expresses that exactly one of `heads` or `tails` is true. It easily desugars to regular probabilistic facts and rules:

```
0.5 :: heads.
tails :- not(heads).
```

ProbLog also allows a rule to be annotated with a probability. This too is merely syntactic sugar, which is desugared into a regular rule and a probabilistic fact.

*PRISM* PRISM follows a different approach. It features a special predicate `msw(i,X)` that introduces an *anonymous* probabilistic choice, also called *switch*. Every new occurrence of an `msw`-predicate during SLD-resolution is seen as a distinct probabilistic choice. The argument `X` is the value that the fact assumes. The range and probability are associated with the identifier argument `i` and provided separately with the predicate `values_x(i,[v1,...,vn],[p1,...,pn])` (green code denotes PRISM code).

*ProbLog vs. PRISM* In what follows, we use the following running example, expressed in PRISM as:[1]

[1] Strictly speaking PRISM requires that probabilistic predicates have at least one (possibly dummy) argument, for the purpose of tabling; we ignore this requirement for simplicity.

```
values_x(i,[t,f],[0.5,0.5]).
p :- q,r.
q :- msw(i,t).
r :- msw(i,t).
```

Here the probability of `p` is 25%. Since there are two choices with two values each, there are four possible combinations, each with probability 25%, but in only one both `q` and `r` are true. The corresponding ProbLog program is:

```
0.5 :: q.
0.5 :: r.
p :- q,r.
```

Again the probability of `p` is 25%.

Note that the following ProbLog program is not equivalent to the PRISM code:

```
0.5 :: q.
p :- q,q.
```

The reason is that ProbLog treats multiple occurrences of identical atoms as a single probabilistic fact (this is called *stochastic memoisation*). Hence the probability of `p` is 50%, since `q` is either true or false with probability 50%. Contrast this with PRISM, where each `msw` is its own random variable, as can be seen from the previous example, where the probability was only 25%.

## 3 Translating PRISM to ProbLog

This section shows how to translate PRISM programs to ProbLog.

### 3.1 Naive Approach

Consider the translation of the following PRISM program, a simplification of our running example, where the probability of `p` is 25%:

```
values_x(i,[t,f],[0.5,0.5]).
p :- msw(i,t),msw(i,t).
```

We could try to simulate the `msw`-predicate in ProbLog with the following annotated disjunction:

```
0.5 :: msw(i,t); 0.5 :: msw(i,f).
p :- msw(i,t),msw(i,t).
```

Unfortunately, this naive transformation is not faithful: there are only two total choices, one where `msw(i,t)` is true and `msw(i,f)` is false, and vice versa. Both have 50% probability, and `p` only holds in the first. The issue is that for ProbLog both occurrences of `msw` refer to the same probabilistic fact that always assumes the same value, while in PRISM they are considered distinct facts that can take different values.

### 3.2 Labeling Approach

In order to distinguish different references to the same `msw` fact in ProbLog, we extend the `msw` predicate with a third argument that uniquely labels each reference. We develop this labelling strategy in several steps, motivating each step with an example.

*Labeling* First, the annotated disjunction for `msw` is extended with a third argument:

```
0.5 :: msw(i,t,_) ; 0.5 :: msw(i,f,_).
```

The disjunction does not actually care what the value of this "label" argument is; its sole purpose is to distinguish different references to the fact.

From the previous example it is clear that different references to the fact should have a different label value. This is accomplished by using the goal position as an identifier. More precisely: every `msw` is labeled with $\mathtt{g}_i$, where $i$ is the position in the body of the clause:

```
0.5 :: msw(i,t,_) ; 0.5 :: msw(i,f,_).
p :- msw(i,t,g1),msw(i,f,g2).
```

The goal `p` now succeeds with probability 25%.

*Clause Labelling* The above labeling is too simplistic to work in general: it does not distinguish calls that occur in the same position of different clauses, or branches of a disjunction. The following PRISM program illustrates the problem.

```
values_x(i,[t,f],[0.5,0.5]).        0.5:: msw(i,t,_); 0.5:: msw(i,f,_).
p :- msw(i,X),q(X).                 p :- msw(i,X,g1),q(X).
q(t).                          ↦    q(t).
q(f) :- msw(i,f).                   q(f) :- msw(i,f,g1).
```

Both switches in the program above map to the same fact in ProbLog, since they share the same $\mathtt{g}_1$ label. In ProbLog, `p` succeeds with probability 100%, while the probability that PRISM computes is 75%. Clearly we need to label switches not just with their position within a clause, but also with the clause that they occur in. The result is the following ProbLog program:

```
p :- msw(i,X,g1(c1)),q(X).
q(t).
q(f) :- msw(i,t,g1(c3)).
```

Now the computed probability is indeed 75%.

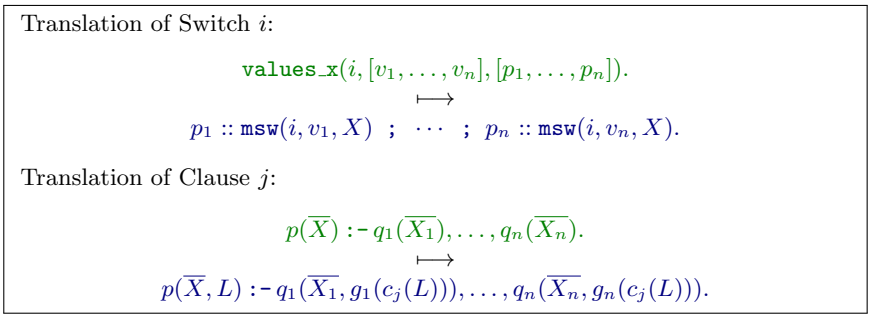


**Fig. 1.** PRISM to ProbLog translation.

*Context Labelling* Finally, the context also matters: consider the following code:

```
values_x(i,[t,f],[0.5,0.5]).          0.5 :: msw(i,t,_) ; 0.5 :: msw(i,f,_).
p :- q,q.                        ↦    p :- q,q.
q :- msw(i,t).                        q :- msw(i,t,g1(c2)).
```

The labelling does not distinguish between the switches that are called due to the first and second occurrence of `q` in `p`. Hence, ProbLog determines that `p` holds with 50% probability, where PRISM computes a 25% probability.

Our solution is to include the label of the parent goal, and by extension that of all ancestors, in the label. In other words, the label becomes a kind of stack trace that documents the path from the top-level goal to the particular invocation of the `msw` fact. To pass this trace around we extend every predicate with a label argument:

```
0.5 :: msw(i,t,_) ; 0.5 :: msw(i,f,_).
p(Label) :- q(g1(c1(Label))),q(g2(c1(Label)).
q(Label) :- msw(i,t,g1(c2(Label))).
```

This ensures that the labels of the two switches are distinct, allowing ProbLog to compute the intended probability. In effect, we accurately model the following requirement from the PRISM manual [1, Section 2.2,p. 17]:

> *The truth-values of switch instances at the different positions of a proof tree are independently assigned. This means that the predicate calls of `msw/2` behave independently of each other.*

*Summary* Figure 1 summarises the translation. Switches are translated to annotated disjunctions of the `msw`/3 predicate, and all predicates receive a label argument that identifies the execution trace up to the call to predicate.

Note that PRISM makes a number of assumptions [1, Section 2.4.6,p. 27] about programs, such as the *exclusiveness condition* which states that branches in a disjunction must be mutually exclusive. The meaning of programs that violate these assumptions is not defined. Hence we also do not define the behavior of the ProbLog translation of undefined PRISM programs.

## 4 Translating ProbLog to PRISM

The transformation from ProbLog to PRISM is more involved, because ProbLog allows richer expressions for its facts (including non-ground arguments), and ProbLog's memoisation of probabilistic facts must be simulated. [2]

### 4.1 Naive Approach

First we consider a naive approach: assume that the ProbLog program has a finite number of probabilistic facts $f_1, \ldots, f_n$. We can choose the values of these facts up front, and modify the remainder of the program to pass these values around. Consider the following example:

```
0.5 :: f1.
0.4 :: f2.
p :- f1, f2.
query(p).
```

For the two probabilistic ProbLog facts `f1` and `f2` we get the PRISM range `[t,f]` for `t`rue and `f`alse and the corresponding probability distributions.

```
values_x(f1,[t,f],[0.5,0.5]).
values_x(f2,[t,f],[0.4,0.6]).
```

The translated `query` predicate chooses the truth values of the two probabilistic facts, and passes the list of true facts to predicate `p`.

```
query :- choose_fact(f1,[],F1), choose_fact(f2,F1,Fs), p(Fs).
```

Here, the PRISM predicate `choose_fact` chooses the truth value for the given fact with `msw`, and, if true, adds it to the given list (see Figure 2).

Finally, the program's clauses are modified to pass the list of true facts around, and invocations of the facts are realised by `member/2` lookups in the list.

```
p(Fs) :- f1(Fs), f2(Fs).
f1(Fs) :- member(f1,Fs).
f2(Fs) :- member(f2,Fs).
```

Figure 2 summarises the general translation scheme. All predicates are extended with an argument for the list of true facts. Probabilistic facts are translated into a lookup in the list of true facts, and each fact receives an associated associated distribution of true and false. Finally, the query chooses up front the truth of the facts and passes the list true facts to the actual query.

[2] In theory, there is a predicate `msw(i,N,V)`, where $n$ distinguishes different instances of the switch $i$, such that different calls to $i$ with unifying $N$ always have the same value $V$, like stochastic memoisation. In practice, `msw/3` is not implemented [8].

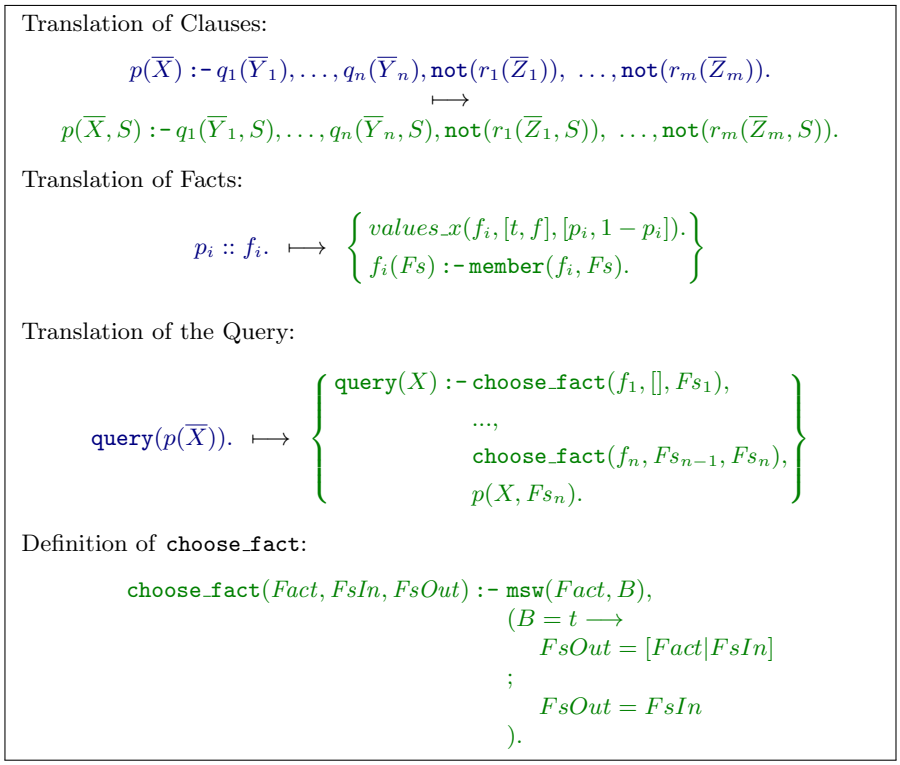


**Fig. 2.** The naive up-front transformation from ProbLog to PRISM.

This transformation closely follows the distribution semantics of ProbLog programs given in Section 2: the `choose_fact` calls compute all possible total choices through backtracking, and PRISM counts the probabilities of those choices for which the query succeeds.

Unfortunately there are a number of problems with this approach. Firstly, the set of probabilistic facts has to be known statically. This is clearly problematic for programs whose grounding contains infinitely many facts (even if answering a query involves only a finite subset). Secondly, even if the program contains only a finite number of facts, the naive translation explores an exponential number of total choices, even if the truth value of many facts in those choices is immaterial.

### 4.2 Dynamic Approach

This section relaxes the requirement that all facts must be known statically with a more dynamic approach. This approach maintains a partial choice, (i.e., a choice where some facts haven not yet been assigned a value) and extends it dynamically whenever an unknown fact is is encountered during the evaluation of the program. At first glance, it seems that this can be accomplished by

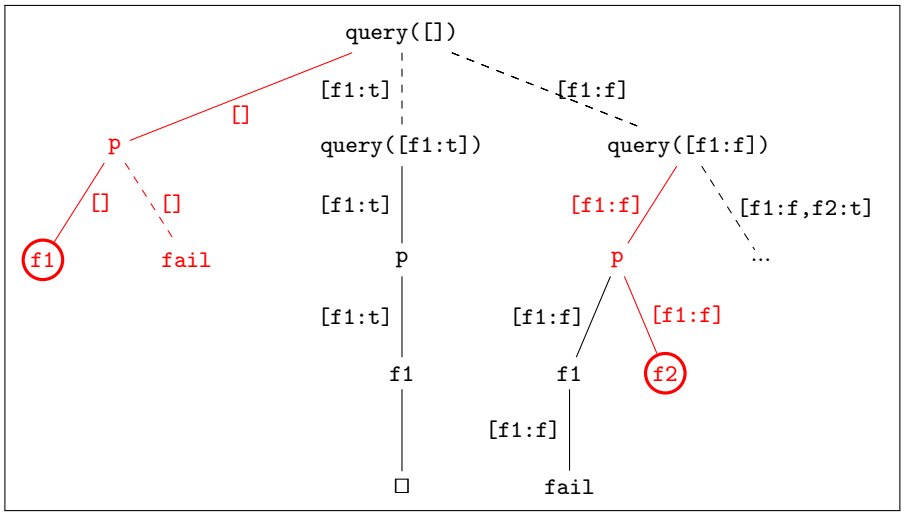


**Fig. 3.** A partial tree showing the execution of the dynamic translation.

maintaining separate input and output lists of facts, and adding new facts to the output list. However, this violates PRISM's exclusiveness condition. Instead, communicating the discovery of as yet unknown facts requires more advanced control flow manipulation, which is nicely captured by Prolog's exception mechanism.[3] The program then simply throws an exception whenever it encounters an unknown fact. The exception raised by the occurence of an unknown fact is handled by probabilistically choosing a value for the fact, and extending the partial choice accordingly. The query is then restarted with this new partial choice. When the query terminates (succeeds or fails), PRISM backtracks over the probabilistic switches, choosing new values for the facts. In this manner, all total choices are eventually explored.

Unfortunately exceptions are incompatible with PRISM's use of tabling. Nevertheless we use them to explain our translation because it nicely captures the idea of the control flow used in our translation. The appendix contains the actual working implementation, encoding the exception control flow in terms of `assert/1` and `retract/1`.

*Translation by Example* It is instructive to look in more detail at a ProbLog example:

```
0.5 :: f1.
0.4 :: f2.
p :- f1.
p :- f2.
```

[3] Where `catch(Goal,Ball,Handler)` calls the `Goal`, which may throw an exception with `throw/1`. If an exception is indeed thrown, the goal and any of its alternatives are aborted, the exception is unified with `Ball` and the `Handler` is called.

which translates to the follwing PRISM code:

```
1   values_x(f1,[t,f],[0.5,0.5]).
2   values_x(f2,[t,f],[0.4,0.6]).
3   f1(Pc) :- member(f1:t,Pc), !.
4   f1(Pc) :- not(member(f1:f,Pc)),throw(unkown(f1)).
5   f2(Pc) :- member(f2:t,Pc), !.
6   f2(Pc) :- not(member(f2:f,Pc)),throw(unkown(f2)).
7
8   p(Pc) :- f1(Pc).
9   p(Pc) :- f2(Pc).
10
11  query(Pc) :-
12    catch(once(p(Pc)),unknown(F),extend(F,Pc)).
13  extend(F,Pc) :-
14      msw(F,V),
15      query([F:V|T]).
```

- Lines **1-6** deal with probabilistic facts. There are multiple important concepts: (1) In lines 1 and 2, the probabilities and values for facts are defined. (2) Lines **3-4** define the logic for facts. Facts take a single argument `Pc`, the partial choice, represented as a list of known facts with their truth value. If the fact does not appear in the list, it is unknown and an `unknown/1`-exception is thrown.
- Lines **8-9** implement the clauses of the program. They are identical to the original, except that the partial choice are passed around.
- Finally, lines **11-12** implement the overall query logic. The main goal is called with the current partial choice, which is initially empty. There are three possible outcomes:
  1. If the query succeeds, the current partial choice is sufficient and the `once/1` discards any alternative ways to make the query succeed under the same partial choice. This is valid because further ways to make the query succeed do not affect its truth or probability.
  2. If the query fails, this means it fails under the current partial choice and other partial choices can be considered elsewhere (see † below).
  3. If the main predicate depends on an unknown fact, its execution and all its choicepoints are aborted by a `throw/1` call which is intercepted by `catch/3` and `extend/2` is called to recover from the exception. Lines `13-15` define this auxiliary predicate, which extends the current partial choice to include the additional fact. Firstly, it conveys the additional fact to PRISM with an `msw`-call (when PRISM backtracks over the `msw`, it chooses different truth values for the fact, thereby exploring alternative partial choices†), and then recursively calls `query/1` with the extended fact list.

Figure 3 shows a partial execution of this program. The edges have been annotated with the current partial choice at that point in the execution. Dashed

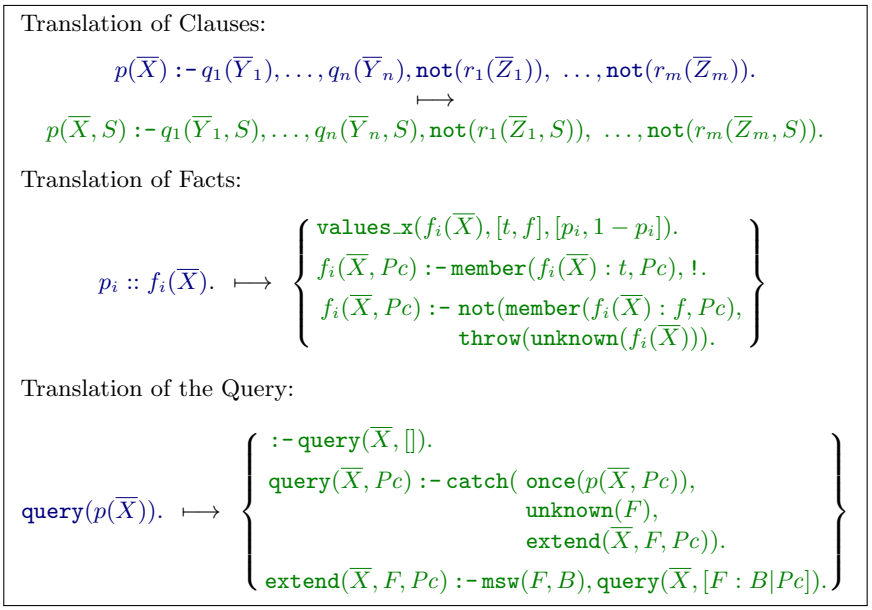


**Fig. 4.** The dynamic transformation from ProbLog to PRISM.

edges represent several derivation steps. Red edges represent the path of an exception. A path of red edges always begins at a node that has been circled in red, the unknown fact where the exception originates. Note how an exception always backtracks to the nearest ancestor `query`, where it is dealt with by extending the partial choice.

This example execution demonstrates an advantage of the dynamic approach: once a proof has been found for `p` under partial choice `[f1:t]` (the middle branch), we need not consider any extension of this partial choice. This is safe, because `p` is also true in any extension of `[f1:t]`.

*Summary* The general formulation of the translation is shown in Figure 4. It supports ProbLog facts with arguments, where different instantiations of these arguments represent independent probabilistic choices in their own right. These are naturally supported by our translation.

*Extension: Flexible Probabilities* ProbLog takes the dynamic nature of probabilistic facts one step further by allowing facts whose probability depends on their parameters. Consider the fact `coin(Bias)` for biased coins where `Bias` is the probability of the fact.

```
Bias :: coin(Bias).
```

We can incorporate this into our translation. However, we need to decouple the statically determined range of values, expressed with facts of PRISM's `values/2` predicate,

```
values(coin(Bias),[t,f]).
```

from the dynamically determined probability, for which we introduce the predicate `probability/2`:

```
probability(coin(Bias),Bias).
```

We use the latter when we extend the partial choice, to dynamically set the probability distribution using PRISM's `set_sw/2`.

```
extend(F,Pc) :-
  probability(F,P),
  Q is 1.0 - P,
  set_sw(F,[P,Q]),
  msw(F,B),
  query([F:B|Pc]).
```

The above approach naturally generalises to ProbLog rules of the form

$$P :: \mathtt{p}(\overline{X})\ \mathtt{:-}\ q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n).$$

which we can desugar into a regular rule and a parameterised probabilistic fact.

$$P :: \mathtt{f}(P).$$
$$\mathtt{p}(\overline{X})\ \mathtt{:-}\ q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n), \mathtt{f}(P).$$

## 5 Conclusion and Future Work

This paper has presented two source-to-source transformations. One translates a model written in PRISM to ProbLog, the other travels in the opposite direction. The transformations clearly show where PRISM and ProbLog diverge in their interpretation of the distribution semantics.

There are several possible avenues for future work:

*Missing Features* The translation does not touch upon PRISM's and ProbLog's different facilities for incorporating observations in probabilistic models. In future work the translations presented here can be extended to allow for such observations.

*Formal Correctness* We have not formally shown the translations to be faithful, that is, preserve the semantics of the program. We have only demonstrated them through examples. Preservation proofs are, of course, complicated by the additional infrastructure that ProbLog and PRISM introduce on top of the distribution semantics, and the complex control flow in the ProbLog to PRISM

translation. Nevertheless, a formal proof can vindicate the translation, or bring to light potential errors. *Proof Idea:* Under a suitable semantics for exceptions, show that a program, and its translation have identical logical behaviour, and compute the same set of probabilistic facts. It then follows that their distribution semantics coincide. Finally, the transformed program must satisfy exclusiveness.

*Performance* Aside from correctness, another important aspect is the overhead introduced by the transformation. We have not yet made any effort to quantify this overhead either in concrete timings or asymptotically. Particularly desirable is a *round-trip property*, that shows that running the translations in sequence (e.g. ProbLog to PRISM and back again) does not introduce excessive overhead. Two obstacles to such a round-trip property are readily apparent: first, since the PRISM to ProbLog translation cannot translate the output of the ProbLog-to-PRISM translation, due to its reliance on exceptions. Secondly, ProbLog's inference is #P-Complete in the worst case, as opposed to PRISM's inference, which runs in linear time. Careful analysis is needed tot ensure that ProbLog's running time does not explode in these cases.

*Acknowledgments* We are grateful to Angelika Kimmig, Anton Dries, Wannes Meert and Luc De Raedt, and the anonymous reviewers for their helpful comments. Alexander Vandenbroucke is an SB PhD Fellow at FWO.

## References

1. PRISM manual version 2.2 (Retrieved Jun 11, 2017)
2. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM 23(4), 733–742 (1976)
3. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., Raedt, L.D.: Inference and learning in probabilistic logic programs using weighted boolean formulas. TPLP 15(3), 358–401 (2015), https://doi.org/10.1017/S1471068414000076
4. Kimmig, A.: A Probabilistic Prolog and its Applications. Ph.D. thesis, KU Leuven (November 2010)
5. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
6. Przymusinski, T.C.: Every logic program has a natural stratification and an iterated least fixed point model. In: PODS. pp. 11–21. ACM Press (1989)
7. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: ICLP. pp. 715–729. MIT Press (1995)
8. Sato, T., Kameya, Y.: New advances in logic-based probabilistic modeling by PRISM. In: Probabilistic Inductive Logic Programming. Lecture Notes in Computer Science, vol. 4911, pp. 118–155. Springer (2008)
9. Staton, S.: Commutative semantics for probabilistic programming. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 855–879. Springer (2017)
10. Tolpin, D., van de Meent, J., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican. In: IFL. pp. 6:1–6:12. ACM (2016)

## A ProbLog to PRISM Transformation without `catch/3` and `throw/1`

This section reimplements the translation given in Section 4.2, using `assert/1` and `retract/1`, instead of `throw/1` and `catch/3`. Recall once more the example from Section 4.2. It is now translated as follows:

```
1:  :- dynamic unknown/1.

2:  values_x(f1,[f1:t,f1:f],[0.5,0.5]).
3:  values_x(f2,[f2:t,f2:f],[0.4,0.6]).
4:  f1(Pc) :- member(f1:t,Pc), !.
5:  f1(Pc) :- not(member(f1:f,Pc)),assert(unknown(f1)),fail.
6:  f2(Pc) :- member(f2:t,Pc), !.
7:  f2(Pc) :- not(member(f2:f,Pc)),assert(unknown(f2)),fail.

8:  p(Pc) :- not(unknown(_)),f1(Pc),not(unknown(_)).
9:  p(Pc) :- not(unknown(_)),f2(Pc),not(unknown(_)).

10:  query(Pc) :-
11:   once(p(Pc))
12:   ;
13:   ( unknown(F),
14:     retract(unknown(F)),
15:     extend(F,Pc)
16:   ).
17:  extend(F,Pc) :-
18:    msw(F,V),
19:    query(X,[F:V|Pc]).
```

- Lines `2-7` simulate probabilistic facts. However, where the original translation throws an exception, this translation instead dynamically asserts `unknown_fact/1` (lines **5** and **7**), to indicate that an unknown fact has been detected. The predicate then fails, to simulate an exception.
- Lines `8-9` implement the clauses of the program, they are identical to the original translation, except that the bodies are now guarded by `not(unknown(_))`. They serve two purposes: (1) the check at the beginning of the body prevents backtracking from exploring alternatives, when an unknown fact has been detected; (2) the check at the end of the body ensures that the clause does not inadvertently succeed when an exception is raised inside a fact that occurs inside a negation (causing that fact to fail, and the negation to succeed).
- Finally lines **10-19** implement the main query logic. As before there are three possible outcomes:
  1. The query succeeds, and the behaviour is exactly as before (line **11**).

2. The query fails, and there is no unknown fact (line **13**), this means that the query fails under the current partial choice, and other partial choices can be considered like the original translation.
3. The query fails, and there is an unknown fact (`unknown(F)` succeeds). This event is handled by (1) resetting the exception with `retract/1`(line `14`) and (2) extending the partial choice with `extend/2`, which is unchanged.