Efficient Implementation of an Extended Semantics for Logic Programs with Annotated Disjunctions

Fabrizio Riguzzi[1] and Terrance Swift[2]

[1] ENDIF – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
fabrizio.riguzzi@unife.it
[2] CENTRIA – Universidade Nova de Lisboa
tswift@cs.suysb.edu

Abstract. Logic Programming with Annotated Disjunctions (LPADs) is a formalism for modeling probabilistic information that has recently received increased attention. The LPAD semantics, while being simple and clear, suffers from the requirement of having function free-programs, which is a strong limitation. In this paper we present an extension of the semantics that removes this restriction and allows us to write programs modeling infinite domains, such as Hidden Markov Models. We show that the semantics is well-defined for a large class of programs. Moreover, we present the algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) for computing the probability of queries to programs according to the extended semantics. Tabling and answer subsumption not only ensure the correctness of the algorithm with respect to the semantics but also make it very efficient on programs without function symbols.
PITA has been implemented in XSB and tested on six domains: two with function symbols and four without. The execution times are compared with those of ProbLog, `cplint` and CVE. PITA was almost always able to solve larger problems in a shorter time on both type of domains.

1 Introduction

Many real world domains only can be represented effectively if we are able to model uncertainty. Recently, there has been an increased interest in logic languages representing probabilistic information due to their succesful use in Machine Learning.

Logic Programs with Annotated Disjunction (LPADs) [24] have attracted the attention of various researchers due to their clarity, simplicity, modeling power and ability to model causation. Their semantics is an instance of the distribution semantics [18]: a theory defines a probability distribution over logic programs and the probability of a query is obtained by summing the probabilities of the programs where the query is true. The semantics of LPADs proposed in [24] requires the programs to be function-free, which is a strong requirement ruling out many interesting programs. Thus, we propose a version of the semantics that allows function symbols, along the lines of [18,13].

The new semantics is based on a program transformation technique that not only allows proving the correctness the semantics but also provides an efficient procedure for computing the probability of queries from LPADs. The algorithm "Probabilistic Inference with Tabling and Answer subsumption" (PITA) builds explanations for every subgoal encountered during a derivation of the query. The explanations are compactly represented using Binary Decision Diagrams (BDDs) that also allow an efficient computation of the probability. Since all the explanations for a subgoal must be found, tabling is very useful to store such information. Tabling has already been shown useful for probabilistic logic programming in [7,15,8]. PITA transforms the input LPAD into a normal logic programs in which the subgoals have an extra argument storing a BDD that represents the explanations for its answers. Moreover, we also exploit answer subsumption to combine explanations coming from different clauses.

PITA draws inspiration from [5] that first proposed to use BDDs for computing the probability of queries for the Problog language, a minimalistic probabilistic extension of Prolog, and from [16] that applied BDDs to the more general LPAD syntax. Other approaches for reasoning on LPADs include [15], where SLG resolution is extended by repeatedly branching on disjunctive clauses, and [11], where CVE is presented that transforms an LPAD into an equivalent Bayesian network and then performs inference on the network using the variable elimination algorithm.

PITA was tested on a number of datasets, both with and without function symbols, in order to evaluate its efficiency. The execution times of PITA were compared with those of `cplint` [16], CVE [11] and ProbLog [9]. PITA was able to successfully solve more complex queries than the other algorithms in most cases and it was also almost always faster both on datasets with and without function symbols.

The paper is organized as follows. Section 2 illustrates the syntax and semantics of LPADs. Section 3 discusses the semantics of LPADs with function symbols. Section 4 gives an introduction to BDDs. Section 5 defines dynamic stratification for LPADs. Section 6 briefly recalls tabling and answer subsumption. Section 7 presents PITA and shows its correctness. Section 8 describes the experiments and Section 9 concludes the paper and presents directions for future works.

2 Logic Programs with Annotated Disjunctions

A *Logic Program with Annotated Disjunctions* [24] consists of a finite set of annotated disjunctive clauses of the form $h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, \ldots, b_m$. In such a clause $h_1, \ldots h_n$ are logical atoms, $b_1, \ldots, b_m$ logical literals, and $\{\alpha_1, \ldots, \alpha_n\}$ real numbers in the interval $[0, 1]$ such that $\sum_{j=1}^{n} \alpha_j \leq 1$. $h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n$ is called the *head* and $b_1, \ldots, b_m$ is called the *body*. Note that if $n = 1$ and $\alpha_1 = 1$ a clause corresponds to a normal program clause, sometimes called a *non-disjunctive* clause. If $\sum_{j=1}^{n} \alpha_j < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom $null$ that does not appear in the body of any clause and whose annotation is $1 - \sum_{j=1}^{n} \alpha_j$. For a clause C of the form above, we define $head(C)$ as $\{(h_i : \alpha_i)|1 \leq i \leq n\}$ if $\sum_{i=1}^{n} \alpha_i = 1$ and as $\{(h_i : \alpha_i)|1 \leq i \leq n\} \cup \{(null : 1 - \sum_{i=1}^{n} \alpha_i)\}$ otherwise. Moreover, we define $body(C)$ as $\{b_i|1 \leq i \leq m\}$, $h_i(C)$ as h_i and $\alpha_i(C)$ as α_i.

If the LPAD is ground, a clause represents a probabilistic choice between the non-disjunctive clauses obtained by selecting only one atom in the head. As usual, if the LPAD T is not ground, T can be assigned a meaning by computing its grounding, $ground(T)$. The semantics of LPADs, given in [24], requires the ground program to be finite, so the program must not contain function symbols if it contains variables.

By choosing a head atom for each ground clause of an LPAD we get a normal logic program called a *possible world* of the LPAD (called an *instance* of the LPAD in [24]). A probability distribution is defined over the space of possible worlds by assuming independence between the choices made for each clause.

More specifically, an *atomic choice* is a triple (C, θ, i) where $C \in T$, θ is a substitution that grounds C and $i \in \{1, \ldots, |head(C)|\}$. (C, θ, i) means that, for ground clause $C\theta$, the head $h_i(C)$ was chosen. A set of atomic choices κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices.

A *selection* σ is a composite choice that, for each clause $C\theta$ in $ground(T)$, contains an atomic choice (C, θ, i) in σ. We denote the set of all selections σ of a program T by $\mathcal{S}_T$. The *probability $P(\kappa)$ of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C,\theta,i)\in\kappa} \alpha_i(C)$. A selection σ identifies a normal logic program w_σ defined as follows $w_\sigma = \{(h_i(C)\theta \leftarrow body(C))\theta | (C, \theta, i) \in \sigma\}$. w_σ is called a *possible world* (or simply *world*) of T. $\mathcal{W}_T$ denotes the set of all the possible worlds of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} \alpha_i(C)$.

We consider only *sound* LPADs, in which every possible world has a total model according to the Well-Founded Semantics (WFS) [22]. In this way, the uncertainty is modeled only by means of the disjunctions in the head and not by the features of the semantics. In the following, $w_\sigma \models \phi$ means that the atom ϕ is true in the well-founded model of the program w_σ.

The probability of a ground atom ϕ according to an LPAD T is given by the sum of the probabilities of the possible worlds where the atom is true under the WFS: $P(\phi) = \sum_{\sigma\in\mathcal{S}_T, w_\sigma\models\phi} P(\sigma)$. It is easy to see that P satisfies the axioms of probability.

Example 1. Consider the dependency of sneezing on having the flu or hay fever:

$C_1 = strong\_sneezing(X) : 0.3 \vee moderate\_sneezing(X) : 0.5 \leftarrow flu(X).$
$C_2 = strong\_sneezing(X) : 0.2 \vee moderate\_sneezing(X) : 0.6 \leftarrow hay\_fever(X).$
$C_3 = flu(david).$
$C_4 = hay\_fever(david).$

This program models the fact that sneezing can be caused by flu or hay fever. The query $strong\_sneezing(david)$ is true in 5 of the 9 instances of the program and its probability is

$$P_T(strong\_sneezing(david)) = 0.3{\cdot}0.2 + 0.3{\cdot}0.6 + 0.3{\cdot}0.2 + 0.5{\cdot}0.2 + 0.2{\cdot}0.2 = 0.44$$

Even if we assumed independence between the choices for individual ground clauses, this does not represents a restriction, in the sense that this still allow to represent all the joint distributions of atoms of the Herbrand base that are representable with a Bayesian network over those variables. Details of the proof are omitted for lack of space.

LPADs can be written by the user or learned from data. When written by the user, the best approach is to write each clause so that it models an causal mechanism of the domain and to choose the parameters on the basis of his knowledge of the mechanism.

3 A Semantics for LPADs with Function Symbols

If a non-ground LPAD T contains function symbols, then the semantics given in Section 2 is not well-defined. In this case, each possible world w_σ is the result of an infinite number of choices and the probability $P(w_\sigma)$ of w_σ is 0 since it is given by the product of an infinite number of factors all smaller than 1. Thus, the probability of a formula is 0 as well, since it is a sum of terms all equal to 0.

Therefore a new definition of the LPAD semantics is necessary. We provide such a definition following the approach in [13] for assigning a semantics to ICL programs with function symbols. A similar result can be obtained using [18].

A composite choice κ identifies a set of possible worlds ω_κ that contains all the worlds relative to a selection that is a superset of κ, i.e., $\omega_\kappa = \{w_\sigma | \sigma \in \mathcal{S}_T, \sigma \supseteq \kappa\}$ We define the set of possible worlds associated to a set of composite choices K: $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$

Given a ground atom ϕ, we define the notion of explanation, covering set of composite choices and mutually incompatible set of explanations. A finite composite choice κ is an *explanation* for ϕ if ϕ is true in every world of ω_κ. In Example 1, the composite choice $\{(C_1, \{X/david\}, 1)\}$ is an explanation for $strong\_sneezing(david)$. A set of composite choices K is *covering* with respect to ϕ if every world w_σ in which ϕ is true is such that $w_\sigma \in \omega_K$. In Example 1, the set of composite choices

$$L_1 = \{\{(C_1, \{X/david\}, 1)\}, \{(C_2, \{X/david\}, 1)\}\} \tag{1}$$

is covering for $strong\_sneezing(david)$. Two composite choices κ_1 and κ_2 are *incompatible* if their union is inconsistent, i.e., if there exists a clause C and a substitution θ grounding C such that $(C, \theta, j) \in \kappa_1, (C, \theta, k) \in \kappa_2$ and $j \neq k$. A set K of composite choices is *mutually incompatible* if for all $\kappa_1 \in K, \kappa_2 \in K, \kappa_1 \neq \kappa_2 \Rightarrow \kappa_1$ and κ_2 are incompatible. The set of composite choices

$$\begin{aligned} L_2 = \{&\{(C_1, \{X/david\}, 1), (C_2, \{X/david\}, 2)\},\\ &\{(C_1, \{X/david\}, 1), (C_2, \{X/david\}, 3)\},\\ &\{(C_2, \{X/david\}, 1)\}\} \end{aligned} \tag{2}$$

is mutually incompatible for the theory of Example 1. The following results of [13] hold also for LPADs.

- Given a finite set K of finite composite choices, there exists a finite set K' of mutually incompatible finite composite choices such that $\omega_K = \omega_{K'}$.
- If K_1 and K_2 are both mutually incompatible sets of composite choices such that $\omega_{K_1} = \omega_{K_2}$ then $\sum_{\kappa \in K_1} P(\kappa) = \sum_{\kappa \in K_2} P(\kappa)$

Thus, we can define a unique probability measure $\mu : \Omega_T \to [0, 1]$ where Ω_T is defined as the set of sets of worlds identified by finite sets of finite composite choices: $\Omega_T = \{\omega_K | K \text{ is a finite set of finite composite choices}\}$. It is easy to see that Ω_T is an algebra over $\mathcal{W}_T$. Then μ is defined by $\mu(\omega_K) = \sum_{\kappa \in K'} P(\kappa)$ where K' is a finite set of finite composite choices that is mutually incompatible and such that $\omega_K = \omega_{K'}$. As for ICL, $\langle \mathcal{W}_T, \Omega_T, \mu \rangle$ is a probability space [10].

Definition 1. *The probability of a ground atom ϕ is given by $P(\phi) = \mu(\{w_\sigma | w_\sigma \in \mathcal{W}_T \wedge w_\sigma \models \phi\}$*

Theorem 2 in Section 7 shows that, if T is a sound LPAD with bounded term-size and ϕ is a ground atom, there is a finite set K of explanations of ϕ such that K is covering. Therefore $P(\phi)$ is well-defined.

In the case of Example 1, L_2 shown in equation 2 is a covering set of explanations for $sneezing(david, strong)$ that is mutually incompatible, so

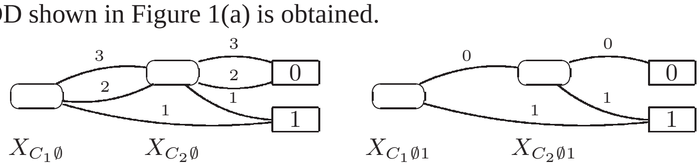
$P(sneezing(david, strong)) = 0.3 \cdot 0.6 + 0.3 \cdot 0.2 + 0.2 = 0.44$

4 Representing Explanations by Means of Decision Diagrams

In order to represent explanations we can use Multivalued Decision Diagrams [21]. An MDD represents a function $f(\mathbf{X})$ taking Boolean values on a set of multivalued variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node has one child for each possible value of the multivalued variable associated to the level of the node. The leaves store either 0 or 1. Given values for all the variables $\mathbf{X}$, an MDD can compute the value of $f(\mathbf{X})$ by traversing the graph starting from the root and returning the value associated to the leaf that is reached.

Given a set of explanations K, we obtain a Boolean function f_K in the following way. Each ground clause $C\theta$ appearing in K is associated to a multivalued variable $X_{C\theta}$ with as many values as atoms in the head of C. Each atomic choice (C, θ, i) is represented by the propositional equation $X_{C\theta} = i$. Equations for a single explanation are conjoined and the conjunctions for the different explanations are disjoined.

The set of explanations in Equation (1) can be represented by the function $f_{L_1}(\mathbf{X}) = (X_{C_1\emptyset} = 1) \vee (X_{C_2\emptyset} = 1)$. An MDD can be obtained from a Boolean function: from f_{L_1} th MDD shown in Figure 1(a) is obtained.



(a) MDD. (b) BDD.

Fig. 1. Decision diagrams for Example 1.

Given a MDD M, we can identify a set of explanations K_M associated to M that is obtained by considering each path from the root to a 1 leaf as an explanation. If is easy to see that K is a set of explanations and M is obtained from f_K, K and K_M represent the same set of worlds, i.e., that $\omega_K = \omega_{K_M}$.

The important role of MDDs is that K_M is mutually incompatible because at each level we branch on a variable and the explanations associated to the leaves that are below a child of a node are incompatible with those of the other children of the node.

By converting a set of explanations into a mutually incompatible set of explanations, MDDs allow to compute $\mu(\omega_K)$ given any K. This is equivalent to computing the probability of a DNF formula which is an NP-hard problem but decision diagrams offer also a practical algorithm that was shown better than other methods [5].

Decision diagrams can be built with various software packages that provide highly efficient implementation of Boolean operations. However, most packages are restricted to work on Binary Decision Diagram (BDD), i.e., decision diagrams where all the variables are Boolean. To work on MDD with a BDD package, we must represent multivalued variables by means of binary variables. Various options are possible, we found that the following, proposed in [4], gives the best performance. For a variable X_1 having n values, we use $n-1$ Boolean variables $X_{11}, \ldots, X_{1n-1}$ and we represent the equation $X_1 = i$ for $i = 1, \ldots n-1$ by means of the conjunction $\overline{X_{11}} \wedge \overline{X_{12}} \wedge \ldots \wedge \overline{X_{1i-1}} \wedge X_{1i}$, and the equation $X_1 = n$ by means of the conjunction $\overline{X_{11}} \wedge \overline{X_{12}} \wedge \ldots \wedge \overline{X_{1n-1}}$. The BDD representation of the function f_{L_1} is given in Figure 1(b). The Boolean variables are associated with the following parameters: $P(X_{11}) = P(X_1 = 1) \ldots P(X_{1i}) = \frac{P(X_1=i)}{\prod_{j=1}^{i-1}(1-P(X_{1i-1}))}$.

5 Dynamic Stratification of LPADs

One of the most important formulations of stratification is that of *dynamic* stratification. [14] shows that a program has a 2-valued well-founded model iff it is dynamically stratified, so that it is the weakest notion of stratification that is consistent with the WFS. As presented in [14], dynamic stratification computes strata via operators on *3-valued interpretations* – pairs of the form $\langle T; F\rangle$, where T and F are subsets of the Herbrand base H_P of a normal program P.

Definition 2. *For a normal program P, sets T and F of ground atoms, and a 3-valued interpretation I we define*

$True_I(T) = \{A : val_I(A) \neq \mathtt{t}$ *and there is a clause* $B \leftarrow L_1, ..., L_n$ *in* P *and a ground substitution* θ *such that* $A = B\theta$ *and for every* $1 \leq i \leq n$ *either* $L_i\theta$ *is true in* I*, or* $L_i\theta \in T\}$*;*

$False_I(F) = \{A : val_I(A) \neq \mathtt{f}$ *and for every clause* $B \leftarrow L_1, ..., L_n$ *in* P *and ground substitution* θ *such that* $A = B\theta$ *there is some* i $(1 \leq i \leq n)$*, such that* $L_i\theta$ *is false in* I *or* $L_i\theta \in F\}$.

The conditions $val_I(A) \neq \mathtt{t}$ and $val_I(A) \neq \mathtt{f}$ are inessential, but ensure that only *new* facts are included in $True_I(T)$ and $False_I(F)$, and simplify the definition of dynamic strata below. [14] shows that $True_I$ and $False_I$ are both monotonic and defines $\mathcal{T}_I$ as the least fixed point of $True_I$ and $\mathcal{F}_I$ as the greatest fixed point of $False_I$. In words, the operator $\mathcal{T}_I$ extends the interpretation I to add the new atomic facts that can be derived from P knowing I; $\mathcal{F}_I$ adds the new negations of atomic facts that can be shown false in P by knowing I (via the uncovering of unfounded sets). An iterated fixed point operator builds up dynamic strata by constructing successive partial interpretations as follows

Definition 3 (Iterated Fixed Point and Dynamic Strata). *For a program P let*

$$\begin{aligned} WFM_0 &= \langle \emptyset; \emptyset \rangle; \\ WFM_{\alpha+1} &= WFM_\alpha \cup \langle \mathcal{T}_{WFM_\alpha}; \mathcal{F}_{WFM_\alpha} \rangle; \\ WFM_\alpha &= \bigcup_{\beta<\alpha} WFM_\beta, \text{ for limit ordinal } \alpha. \end{aligned}$$

Let $WFM(P)$ denote the fixed point interpretation WFM_δ, where δ is the smallest countable ordinal such that both sets $\mathcal{T}_{WFM_\delta}$ and $\mathcal{F}_{WFM_\delta}$ are empty. We refer to δ as the depth *of program P. The* stratum *of atom A, is the least ordinal β such that $A \in WFM_\beta$ (where A may be either in the true or false component of WFM_β).*

[14] shows that the iterated fixed point $WFM(P)$ is in fact the well-founded model and that any undefined atoms of the well-founded model do not belong to any stratum – i.e. they are not added to WFM_δ for any ordinal δ.

Dynamic stratification captures the order in which recursive components of a program must be evaluated. Because of this, dynamic stratification is useful for modeling operational aspects of program evaluation. Fixed-order dynamic stratification [17], used in Section 7, replaces the definition of $False_I(F)$ in Definition 2 is by

$False_I(F) = \{A : val_I(A) \neq \mathtt{f}$ and for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ there exists a ***failing prefix***: i.e., there is some i $(1 \leq i \leq n)$, such that $L_i\theta$ is false in I or $L_i\theta \in F$, and for all j $(1 \leq j \leq i-1)$, $L_j\theta$ is true in $I\}$.

[17] describes how fixed-order dynamic stratification captures those programs that a tabled evaluation can evaluate with a fixed literal selection strategy (i.e. without the SLG operations of SIMPLIFICATION and DELAY). As shown from the following example, fixed-order stratification is a fairly weak condition for a program.

Example 2. The following program has a 2-valued well-founded model and so is dynamically stratified, but does not belong to other stratification classes, such as local, modular, or weak stratification.

$s \leftarrow \neg s, p.$

$p \leftarrow q, \neg r, \neg s.$

$r \leftarrow p, \neg q.$

$s \leftarrow \neg p, \neg q, \neg r.$

$q \leftarrow r, \neg p.$

p, *q*, and *r* all belong to stratum 0, while *s* belongs to stratum 1. The simple program

$p \leftarrow \neg\, p.$ $\quad p.$

is fixed-order stratified, but not locally, modularly, or weakly stratified. Fixed-order stratification is more general than local stratification, and than modular stratification (since modular stratified programs can be decidably rearranged so that they have failing prefixes). It is neither more nor less general than weak stratification.

The above definitions of (fixed-order) dynamic stratification for normal programs can be straightforwardly adapted to LPADs – an LPAD is *(fixed-order) dynamically stratified* if each $w \in \mathcal{W}_\mathcal{T}$ is (fixed-order) dynamically stratified.

6 Tabling and Answer Subsumption

The idea behind tabling is to maintain in a table both subgoals encountered in a query evaluation and answers to these subgoals. If a subgoal is encountered more than once, the evaluation reuses information from the table rather than re-performing resolution against program clauses. Although the idea is simple, it has important consequences. First, tabling ensures termination of programs with the *bounded term size property*. A program P has the bounded term size property if there is a finite function $f : N \rightarrow N$ such that if, a query term Q to P has size $size(Q)$, then no term used in the derivation of Q has size greater than $f(size(Q))$. This makes it easier to reason about termination than in basic Prolog. Second, tabling can be used to evaluate programs with negation according to the WFS. Third, for queries to wide classes of programs, such as datalog programs with negation, tabling can achieve the optimal complexity for query evaluation. And finally, tabling integrates closely with Prolog, so that Prolog's familiar programming environment can be used, and no other language is required to build complete systems. As a result, a number of Prologs now support tabling including XSB, YAP, B-Prolog, ALS, and Ciao. In these systems, a predicate p/n is evaluated using SLDNF by default: the predicate is made to use tabling by a declaration such as *table p/n* that is added by the user or compiler.

This paper makes use of a tabling feature called *answer subsumption*. Most formulations of tabling add an answer A to a table for a subgoal S only if A is a not a variant (as a term) of any other answer for S. However, in many applications it may be useful to order answers according to a partial order or (upper semi-)lattice. In the case of a lattice, answer subsumption may be specified by means of a declaration such as *table p(_,or/3 - zero/1)*. for an unary predicate p. where $zero/1$ is the bottom element of the lattice and $or/3$ is the join operation of the lattice. For example, in the PITA algorithm for LPADs presented in Section 7, if a table had an answer $p(a, E_1)$ and a new answer $p(a, E_2)$ were derived, where E_1 and E_2 are probabilistic explanations, the answer $p(a, E_1)$ is replaced by $p(a, E_3)$, where E_3 is obtained by calling $or(E_1, E_2, E_3)$ and is the logical disjunction of the first two explanations, as stored in a BDD[3]. Answer subsumption over arbitrary upper semi-lattices is implemented in XSB for stratified programs [20]; in addition, the mode-directed tabling of B-Prolog can also be seen as a form of answer subsumption.

Section 7 uses SLG resolution [3] extended with answer subsumption in its proof of Theorem 2, although similar results could be extended to other tabling formalisms that support negation and answer subsumption.

7 Program Transformation

The first step of the PITA algorithm is to apply a program transformation to an LPAD to create a normal normal program that contains calls for manipulating BDDs. In our implementation, these calls provide a Prolog interface to the CUDD[4] C library and use the following predicates[5]

– *init, end*: for allocation and deallocation of a BDD manager, a data structure used to keep track of the memory for storing BDD nodes;

[3] The logical disjunction E_3 can be seen as subsuming E_1 and E_2 over the partial order af implication defined on logical formulas.

[4] `http://vlsi.colorado.edu/~fabio/`

[5] BDDs are represented in CUDD as pointers to their root node.

- *zero(-BDD), one(-BDD), and(+BDD1,+BDD2,-BDDO), or(+BDD1,+BDD2, -BDDO), not(+BDDI,-BDDO)*: Boolean operations between BDDs;
- *add_var(+N_Val,+Probs,-Var)*: addition of a new multi-valued variable with *N_Val* values and parameters *Probs*;
- *equality(+Var,+Value,-BDD)*: *BDD* represents *Var=Value*, i.e. that the random variable *Var* is assigned *Value* in the BDD;
- *ret_prob(+BDD,-P)*: returns the probability of the formula encoded by *BDD*.

add_var(+N_Val,+Probs,-Var) adds a new random variable associated to a new instantiation of a rule with *N_Val* head atoms and parameters list *Probs*. The auxiliary predicate *get_var_n(+R,+S,+Probs,-Var* is used to wrap *add_var/3* and avoid adding a new variable when one already exists for an instantiation. As shown below, a new fact *var(R,S,Var)* is asserted each time a new random variable is created, where *R* is an identifier for the LPAD clause, *S* is a list of constants, one for each variables of the clause, and *Var* is a integer that identifies the random variable associated with clause *R* under a particular grounding. The auxiliary predicates has the following definition

$get\_var\_n(R, S, Probs, Var) \leftarrow$
$(var(R, S, Var) \rightarrow true;$
$length(Probs, L), add\_var(L, Probs, Var), assert(var(R, S, Var))).$

where *Probs* is a list of floats that stores the parameters in the head of rule *R*.

The PITA transformation applies to clauses, literals and atoms. If h is an atom, $PITA_h(h)$ is h with the variable BDD added as the last argument. If b_j is an atom, $PITA_b(b_j)$ is b_j with the variable B_j added as the last argument. In either case for an atom a, $BDD(PITA(a))$ is the value of the last argument of $PITA(a)$,

If b_j is negative literal $\neg a_j$, $PITA_b(b_j)$ is the conditional $(PITA'_b(a_j) \rightarrow not(BN_j, B_j); one(B_j))$, where $PITA'_b(a_j)$ is a_j with the variable BN_j added as the last argument. In other words the input BDD, BN_k, is negated if it exists; otherwise the BDD for the constant function 1 is returned.

A non-disjunctive fact $C_r = h$ is transformed into the clause
$PITA(C_r) = PITA_h(h) \leftarrow one(BDD)$.
A disjunctive fact $C_r = h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n$. where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$[6]

$$PITA(C_r, 1) = PITA_h(h_1) \leftarrow get\_var\_n(i, [], [\alpha_1, \ldots, \alpha_n], Var), equality(Var, 1, BDD).$$
$$\ldots$$
$$PITA(C_r, n) = PITA_h(h_n) \leftarrow get\_var\_n(r, [], [\alpha_1, \ldots, \alpha_n], Var), equality(Var, n, BDD).$$

In the case where the parameters do not sum to one, the clause is first transformed into $h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \vee null : 1 - \sum_1^n \alpha_i$. and then into the clauses above, where the list of parameters is $[\alpha_1, \ldots, \alpha_n, 1 - \sum_1^n \alpha_i]$ but the $(n+1)$-th clause (the one for $null$) is not generated.

The definite clause $C_r = h \leftarrow b_1, b_2, \ldots, b_m$. is transformed into the clause

$$PITA(C_r) = PITA_h(h) \leftarrow PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \ldots, PITA_b(b_m), and(BB_{m-1}, B_m, BDD).$$

[6] The second argument of get_var_n is the empty list because, since we are considering only range restricted programs (cfr. below), a fact does not contain variables.

The disjunctive clause

$C_r = h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, b_2, \ldots, b_m.$

where the parameters sum to 1, is transformed into the set of clauses $PITA(C_r)$

$$\begin{array}{ll} PITA(C_r, 1) = PITA_h(h_1) \leftarrow & PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \\ & \ldots, PITA_b(b_m), and(BB_{m-1}, B_m, BB_m), \\ & get\_var\_n(r, VC, [\alpha_1, \ldots, \alpha_n], Var), \\ & equality(Var, 1, B), and(BB_m, B, BDD). \\ \ldots & \\ PITA(C_r, n) = PITA_h(h_n) \leftarrow & PITA_b(b_1), PITA_b(b_2), and(B_1, B_2, BB_2), \\ & \ldots, PITA_b(b_m), and(BB_{m-1}, B_m, BB_m), \\ & get\_var\_n(r, VC, [\alpha_1, \ldots, \alpha_n], Var), \\ & equality(Var, n, B), and(BB_m, B, BDD). \end{array}$$

where VC is a list containing each variable appearing in C_r. If the parameters do not sum to 1, the same technique used for disjunctive facts is used.

Example 3. Clause C_1 from the LPAD of Example 1 is translated into

$$\begin{array}{ll} strong\_sneezing(X, BDD) & \leftarrow flu(X, B_1), \\ & get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var), \\ & equality(Var, 1, B), and(B_1, B, BDD). \\ moderate\_sneezing(X, BDD) & \leftarrow flu(X, B_1), \\ & get\_var\_n(1, [X], [0.3, 0.5, 0.2], Var), \\ & equality(Var, 2, B), and(B_1, B, BDD). \end{array}$$

while clause C_3 is translated into

$flu(david, BDD) \leftarrow one(BDD).$

In order to answer queries, the goal *solve(Goal,P)* is used, which is defined by

$$\begin{array}{ll} solve(Goal, P) \leftarrow & init, retractall(var(\_, \_, \_)), \\ & add\_bdd\_arg(Goal, BDD, GoalBDD), \\ & (call(GoalBDD) \rightarrow ret\_prob(BDD, P); P = 0.0), \\ & end. \end{array}$$

where $add\_bdd\_arg(Goal, BDD, GoalBDD)$ implements $PITA_h(Goal)$. Moreover, various predicates of the LPAD should be declared as tabled. For a predicate p/n, the declaration is *table p(_1,...,_n,or/3-zero/1)*, that indicates that answer subsumption is used to form the disjunct of multiple explanations: At a minimum, the predicate of the goal should be tabled; as shown in Section 8 it is usually better to table every predicate whose answers have multiple explanations and are going to be reued often.

Correctness of PITA In this section we show two results regarding the PITA transformation and its tabled evaluation[7]. These results ensure on one hand that the semantics is well-defined and on the other hand that the evaluation algorithm is correct. For the purposes of our semantics, we consider the BDDs produced as ground terms, and do not specify them further. We first state the correctness of the PITA transformation with respect to the well-founded semantics of LPADs. Because we allow LPADs to have

[7] Due to space limitations, our presentation is somewhat informal: a formal presentation with all proofs and supporting definitions can be found at `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigSwi10-TR.pdf`.

function symbols, care must be taken to ensure that explanations are finite. To accomplish this, we prove correctness for what we term dynamically-finitary programs, essentially those for which a derivation in the well-founded semantics does not depend on an infinite unfounded set [8].

Theorem 1 (Correctness of PITA Transformation). Let T be a sound dynamically-finitary LPAD. Then κ is an explanation for a ground atom a iff there is A $PITA_h(a)\theta$ in $WFM(\ PITA(ground(T)))$, such that κ is a path in $BDD(PITA_h(a)\theta)$ to a 1 leaf.

Theorem 2 below states the correctness of the tabling implementation of PITA, since the BDD returned for a tabled query is the disjunction of a set of covering explanations for that query. The proof uses an extension of SLG evaluation that includes answer subsumption but that is restricted to fixed-order dynamically stratified programs [20], a formalism that models the implementation tested in Section 8. Note that unlike Theorem 1, Theorem 2 does not require the program T to be grounded. However, Theorem 2 does require T to be range restricted in order to ensure that tabled evaluation grounds answers. A normal program/LPAD is *range restricted* if all the variables appearing in the head of each clause appear also in the body. If a normal program is range restricted, every successful SLDNF-derivation for G completely grounds G [12], a result that can be straightforwardly extended to tabled evaluations. In addition, Theorem 2 requires T to have the bounded term-size property (cf. Section 6) to ensure termination and finite explanations.

Theorem 2 (Correctness of PITA Evaluation). Let T be a range restricted, bounded term-size, fixed-order dynamically stratified LPAD and a a ground atom. Let $\mathcal{E}$ be an SLG evaluation of $PITA_h(a)$ against $PITA(T)$, such that answer subsumption is declared on $PITA_h(a)$ using BDD-disjunction. Then $\mathcal{E}$ terminates with an answer ans for $PITA_h(a)$ and $BDD(ans)$ represents a covering set of explanations for a.

Thus range restricted, bounded term-size and fixed-order dynamically stratified LPADs have a finite set of explanations that are covering for a ground atom, so the semantics with function symbols is well-defined.

8 Experiments

PITA was tested on two datasets that contain function symbols: the first is taken from [24] and encodes a Hidden Markov Model (HMM) while the latter from [5] encodes biological networks. Moreover, it was also tested on the four testbeds of [11] that do not contain function symbols. PITA was compared with the exact version of ProbLog[9] [5] available in the git version of Yap as of 19/12/2009, with the version of `cplint`

[8] Dynamically-finitary programs are a strict superclass of the finitary programs of [1] and are neither a subclass nor a superclass of the finitely ground programs of [2]. The formal definition of dynamically-finitary programs is in the full version of this paper.

[9] ProbLog was not tested on programs with more than two atoms in the head because the publicly available version is not yet able to deal with non-binary variables.

[16] available in Yap 6.0 and with the version of CVE [11] available in ACE-ilProlog 1.2.20[10].

The first problem models a HMM with three states 1, 2 and 3 of which 3 is an end state. This problem is encoded by the program

s(0,1):1/3 $\vee$ *s(0,2):1/3* $\vee$ *s(0,3):1/3.*

s(T,1):1/3 $\vee$ *s(T,2):1/3* $\vee$ *s(T,3):1/3* $\leftarrow$ *T1 is T-1, T1>=0, s(T1,F),* \+ *s(T1,3)*

For this experiment, we query the probability of the HMM being in state 1 at time *N* for increasing values of *N*, i.e., we query the probability of *s(N,1)*. In PITA, we did not use reordering of BDDs variables[11]. The execution times of PITA, CVE and `cplint` are shown in Figure 2. In this problem tabling provides an impressive speedup, since computations can be reused often.

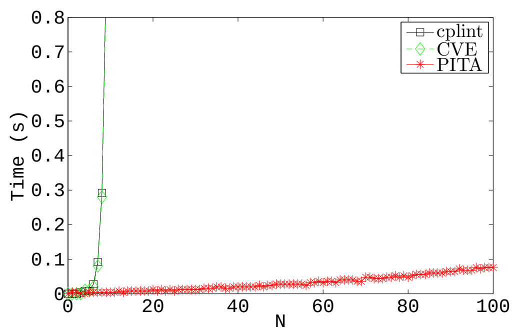


Fig. 2. Three sided die.

The biological network problems compute the probability of a path in a large graph in which the nodes encode biological entities and the links represents conceptual relations among them. Each programs in this dataset contains a definition of path plus a number of links represented by probabilistic facts. The programs have been sampled from a very large graph and contain 200, 400, . . ., 5000 edges. Sampling was repeated ten times, to obtain a series of 10 programs of increasing size. In each test we queried the probability that the two genes HGNC_620 and HGNC_983 are related. We used the definition of path of [9] that performs loop checking explicitly by keeping the list of visited nodes:

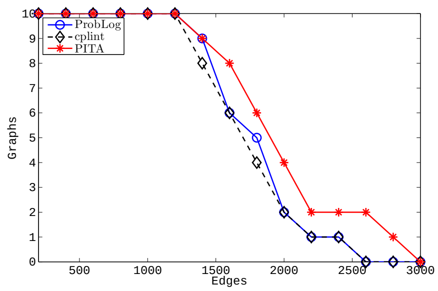
$$\begin{array}{ll} path(X,Y) & \leftarrow path(X,Y,[X],Z). \\ path(X,Y,V,[Y|V]) & \leftarrow edge(X,Y). \\ path(X,Y,V0,V1) & \leftarrow edge(X,Z), append(V0,\_S,V1), \\ & \quad \backslash + member(Z,V0), path(Z,Y,[Z|V0],V1). \end{array}$$

We used this definition because it gave better results than the one without explicit loop checking. The possibility of using lists (that require function symbols) allowed in this case more modeling freedom. The predicates $path/2$ and $edge/2$ are tabled.

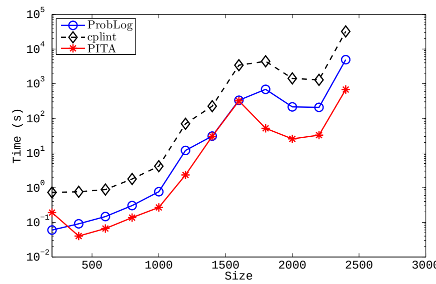
We ran PITA, ProbLog and `cplint` on the graphs in sequence starting from the smallest program and in each case we stopped after one day or at the first graph for

[10] All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

[11] For each experiment, we used either group sift automatic reordering or no reordering of BDDs variables depending on which gave the best results.

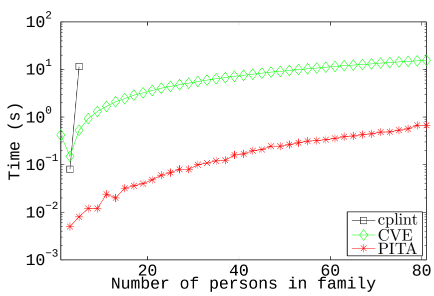


(a) Number of successes.

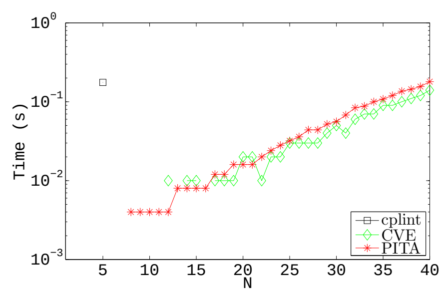


(b) Execution times.

Fig. 3. Biological graph experiments.

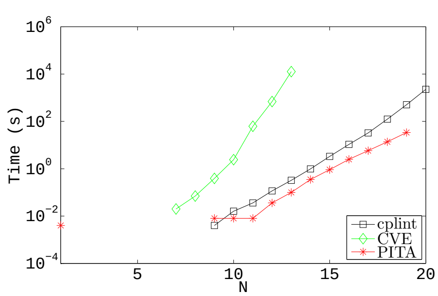


(a) `bloodtype`.

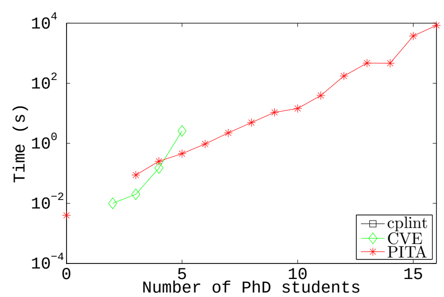


(b) `growingbody`.

Fig. 4. Datasets from (Meert et al. 2009).



(a) `growinghead`.



(b) `uwcse`.

Fig. 5. Datasets from (Meert et al. 2009).

which the program ended for lack of memory[12]. In PITA, we used group sift reordering of BDDs variables. Figure 3(a) shows the number of subgraphs for which each algorithm was able to answer the query as a function of the size of the subgraphs, while Figure 3(b) shows the execution time averaged over all and only the subgraphs for which all the algorithms succeeded. PITA was able to solve more subgraphs and in a shorter time than `cplint` and ProbLog. For PITA the vast majority of time for larger graphs was spent on BDD maintenance. ProbLog ended for lack of memory in three cases out of ten, PITA in two and `cplint` in four. This shows that, even if tabling consumes more memory when finding the explanations, BDDs are built faster and using less memory, probably due to the fact that tabling allows less redundancy (only one BDD is stored for an answer) and a bottom-up construction of the BDDs, which is usually better. This shows that one should table every predicate whose answer have multiple explanations, as $path/2$ and $edge/2$ above.

The four datasets of [11], served as a final suite of benchmarks. `bloodtype` encodes the genetic inheritance of blood type, `growingbody` contains programs with growing bodies, `growinghead` contains programs with growing heads and `uwcse` encodes a university domain. In PITA we disabled automatic reordering of BDDs variables for all datasets except for `uwcse`. The execution times of `cplint`, CVE and PITA are shown in Figures 4(a) and 4(b), 5(a) and 5(b)[13]. PITA was faster than `cplint` in all domains and faster than CVE in all domains except `growingbody`. `growingbody`, however, is a domain in which all the clauses are mutually exclusive, thus making possible to compute the probability even without BDDs.

9 Conclusion and Future Works

This paper has made two contributions. The first, semantic, contribution extends LPADs to include functions. By way of proving correctness of the PITA transformation we also characterize those extended LPAD programs whose derived atoms have only finite explanations (dynamically-finitary LPADs); by way of proving correctness of PITA evaluation we characterize those that have only finite sets of explanations (LPADs with the bounded term-size property). Such results ensure that the semantics with function symbols is well-defined.

The PITA transformation also provides a practical reasoning algorithm that was directly used in the experiments of Section 8. The experiments substantiate the PITA approach. Accordingly PITA programs should be easily portable to other tabling engines such as that of YAP, Ciao and B Prolog if they support answer subsumption over general semi-lattices.

In the future, we plan to extend PITA to the whole class of sound LPADs by implementing the SLG DELAYING and SIMPLIFICATION operations for answer subsumption. In addition, we are developing a version of PITA that is able to answer queries in an approximate way, similarly to [9].

[12] CVE was not applied to this dataset because the current version can not handle graph cycles.

[13] For the missing points at the beginning of the lines a time smaller than 10^{-6} was recorded. For the missing points at the end of the lines the algorithm exhausted the available memory.

References

1. Baselice, S., Bonatti, P., Criscuolo, G.: On finitely recursive programs. Theory and Practice of Logic Programming 9(2), 213–238 (2009)
2. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: ICLP. pp. 407–424 (2008)
3. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM 43(1), 20–74 (1996)
4. De Raedt, L., Demoen, B., Fierens, D., Gutmann, B., Janssens, G., Kimmig, A., Landwehr, N., Mantadelis, T., Meert, W., Rocha, R., Santos Costa, V., Thon, I., Vennekens, J.: Towards digesting the alphabet-soup of statistical relational learning. In: NIPS*2008 Workshop on Probabilistic Programming (2008)
5. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Internation Joint Conference on Artificial Intelligence. pp. 2462–2467 (2007)
6. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming - Theory and Applications, LNCS, vol. 4911. Springer (2008)
7. Kameya, Y., Sato, T.: Efficient EM learning with tabulation for parameterized logic programs. In: Computational Logic. LNCS, vol. 1861, pp. 269–284. Springer (2000)
8. Kimmig, A., Gutmann, B., Santos Costa, V.: Trading memory for answers: Towards tabling ProbLog. In: International Workshop on Statistical Relational Learning. KU Leuven (2009)
9. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of ProbLog programs. In: ICLP. LNCS, vol. 5366, pp. 175–189. Springer (2008)
10. Kolmogorov, A.N.: Foundations of the Theory of Probability. Chelsea Publishing Company, New York (1950)
11. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: International Conference on Inductive Logic Programming. KU LEuven, Leuven, Belgium (2009)
12. Muggleton, S.: Learning stochastic logic programs. Electron. Trans. Artif. Intell. 4(B), 141–153 (2000)
13. Poole, D.: Abducing through negation as failure: stable models within the independent choice logic. J. Log. Program. 44(1-3), 5–35 (2000)
14. Przymusinski, T.: Every logic program has a natural stratification and an iterated least fixed point model. In: Symposium on Principles of Database Systems. pp. 11–21. ACM Press (1989)
15. Riguzzi, F.: Inference with logic programs with annotated disjunctions under the well founded semantics. In: ICLP. pp. 667–771. No. 5366 in LNCS, Springer (2008)
16. Riguzzi, F.: A top down interpreter for LPAD and CP-logic. In: Congress of the Italian Association for Artificial Intelligence. pp. 109–120. No. 4733 in LNAI, Springer (2007)
17. Sagonas, K., Swift, T., Warren, D.S.: The limits of fixed-order computation. Theor. Comput. Sci. 254(1-2), 465–499 (2000)
18. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: ICLP. pp. 715–729 (1995)
19. Swift, T.: A new formulation of tabled resolution with delay. In: Recent Advances in Artifiial Intelligence. LNAI, vol. 1695, pp. 163–177. Sringer (1999)
20. Swift, T.: Tabling for non-monotonic programming. Ann. Math. Artif. Intell. 25(3-4), 201–240 (1999)
21. Thayse, A., Davio, M., Deschamps, J.P.: Optimization of multivalued decision algorithms. In: International Symposium on Multiple-Valued Logic. pp. 171–178 (1978)

22. van Gelder, A., Ross, K., Schlipf, J.: Unfounded sets and well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
23. Vennekens, J., Denecker, M., Bruynooghe, M.: CP-logic: A language of causal probabilistic events and its relation to logic programming. Theory Pract. Log. Program. 9(3), 245–308 (2009)
24. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: ICLP. LNCS, vol. 3131, pp. 195–209. Springer (2004)

A Proof of Theorems 1 and 2

Before proving the theorems, we discuss some of the concepts on which they are based. First, the theorems below use the notion of a dynamically-finitary program.

Definition 4 (dynamically-finitary Programs). *A* dynamically-finitary *normal program P is one in which any atom in $WFM(P)$ (Definition 3) has a finite stratum δ, and for which each stratum $\beta \leq \delta$ was computed by a finite number of applications of the $\mathcal{T}_I$ and $\mathcal{F}_I$ operators (Definition 2).*

An LPAD T is dynamically-finitary if each of its worlds ($w \in \mathcal{W}_T$) is dynamically-finitary.

While the property of being dynamically-finitary is clearly undecidable, note that a dynamically-finitary program P need not have a finite ground instantiation nor a finite depth (cf. Definition 3). However in a dynamically-finitary program, a ground atom a can be seen to be true or false after a finite bottom-up derivation. Since we consider only (subsets of) dynamically stratified programs, the question of determining that an atom is undefined in a dynamically-finitary program does not arise here.

Second, the theorems make explicit mention of BDD data structures. Note that there are a number of ways to represent BDDs as ground terms, although of course a ground term representation of a BDD will be less efficient than that used by CUDD or other packages. While the proofs below all assume that a BDD is a ground term, they do not need to further specify its form. Accordingly the BDD operations used in the PITA transformation: *and/3*, *or/3*, *not/2*, *one/1*, *zero/1*, and *equality/3* are all taken as (infinite) relations on terms, so that these predicates can be made part of a program's ground instantiation in the normal way. As a result, the ground instantiation of $PITA(T)$ instantiates all variables in T with all BDD terms.

Finally, the definition of *get_var_n/4* from Section 7:

$get\_var\_n(RuleName, Vars, Probs, RV) \leftarrow$
$(var(RuleName, Vars, RV) \rightarrow true;$
$length(Probs, L), add\_var(L, Probs, RV),$
$assert(var(RuleName, Vars, RV))).$

uses a non-logical update of the program, and so without modifications, it is not suitable for our proofs below. Alternately, we assume that $ground(T)$ is augmented with a (potentially infinite) number of facts of the form $var(RuleName, [], RV)$ for each ground rule $RuleName$ (note that no variable instantiation is needed in the second argument of $var/3$ if it is indexed on ground rule names). Clearly, the augmentation of T by such facts has the same meaning as *get_var_n/4*, but is simply done by a priori program extension rather than during the computation as in the implementation.

Lemma 1. *If T is a dynamically-finitary LPAD, then $PITA(T)$ is dynamically-finitary, and for each ground atom a, if $PITA_h(a)$ is true in $WFM(PITA(T))$, a has an explanation.*

Proof. If T is dynamically-finitary, each world in $\mathcal{W}_T$ is dynamically-finitary, by Definition 4. By Definition 2 there are interpretations I' and T' such that $a \in True_{I'}(T')$. Since I' an T' were both produced by a finite number of iterations of the $True_I$ and

$False_I$ operators, a "depends" (e.g. in the sense of local stratification) on the truth value of a finite set $\mathcal{S}$ of atoms. Since any world in which $\mathcal{S}$ is true implies that a is true, $\mathcal{S}$ is an explanation for a.

By the preceding discussion, the extra literals introduced into the bodies of rules by the $PITA$ transformation are all facts. Accordingly, for e.g. $and/3$, $and/3$ atoms that are in the $and/3$ relation are true at all points in the well-founded computation after $True_\emptyset(\emptyset)$, and all $and/3$ atoms that are not in the $and/3$ relation are false at all points after $False_\emptyset(\emptyset)$. Accordingly, derivation of an atom $PITA_h(a)$ in $PITA(T)$ will take a finite number of steps (i.e. applications of the $True$, $False$ and WFM operators) if the derivation of a takes a finite number of steps in each possible world of T, so that $PITA(T)$ is dynamically-finitary.

Theorem 1 *Let T be a sound dynamically-finitary LPAD. Then κ is an explanation for a ground atom a iff there is some $PITA_h(a)\theta$ in $WFM(PITA(ground(T)))$, such that κ is a path to a 1 leaf in $BDD(PITA_h(a)\theta)$*[14].

Proof. (Soundness $\Leftarrow$) The proof is by outer induction on the operator WFM (Definition 3) and inner induction on the operators $True_I$ and $False_I$ (Definition 2). Within each stratum there is induction on the structure of formulas. We note that since T is dynamically-finitary, we do not need to consider transfinite induction.

1. *Base Case for Outer Induction (WFM Operator)*. This case consists of induction for $True_\emptyset$ and $False_\emptyset$
   (a) *Base Case for Inner Induction ($True_\emptyset$ and $False_\emptyset$)* In the case of $True_\emptyset$, a ground atom a is proven using a non-disjunctive fact or disjunctive fact in the T. If a unifies with a non-disjunctive fact f,

   $$PITA(f) = PITA_h(f) \leftarrow one(BDD).$$

   so the BDD associated with a represents the constant 1, i.e., an empty composite choice that is an explanation for a. Alternately, if a unifies with head $h_i(C_r)$ of a disjunctive fact C_r,

   $$PITA(C_r, i) = PITA_h(h_1(C_r)) \leftarrow var(C_r, [], Var), equality(Var, i, BDD).$$

   the rule "creates" a new variable Var and a new BDD BDD indicating that the value of Var is i to represent the atomic choice $\{(C_r, \emptyset, i)\}$, which is an explanation for a.
   In the base case of $False_\emptyset$, those atoms *removed* from the greatest fixed point are non-disjunctive facts contained in $ground(T)$.
   (b) *Inductive Case (Inner)* In the case of $True_\emptyset$, the atom a is proven via a rule C_r in $PITA(T)$ that does not contain negation in its body. If C_r is non-disjunctive

   $$\begin{aligned} PITA(C_r) = PITA_h(h) \leftarrow\ & PITA_b(b_1), PITA_b(b_2), \\ & and(B_1, B_2, BB_2), \ldots, \\ & PITA_b(b_m), and(BB_{m-1}, B_m, BDD). \end{aligned}$$

[14] The θ substitution binds the variable introduced by $PITA_h(a)$ to a BDD-term.

By the induction assumption, the statement holds for $PITA_b(b_i) : 1 \leq i \leq m$. For $j : 1 \leq j < m$, let BBj be the BDD assocated with the $\bigwedge_{1 \leq i \leq j} PITA_b(b_i)$. Then the literal $and(BB_j, B_{j+1}, BB_{j+1})$ produces the BDD BB_{j+1} that is either the first argument of the next $and/3$ literal in the body, or is the final BDD produced in the body. We now perform structural induction on the body of C_r. For the base case, each path in B_1 is an explanation by the inner fixed point inductive assumption. For the inductive case, by the structural induction assumption, BB_k represents a finite set of explanations for $\bigwedge_{1 \leq i \leq k} PITA_b(b_i)$, and B_{k+1} for $PITA_b(b_{k+1})$ by the inner fixed point inductive case, so that clearly $and(BB_k, B_{k+1}), BB_{k+1})$ produces a finite set of explanations for $\bigwedge_{1 \leq i \leq k+1} PITA_b(b_i)$.

Alternately, if a is derived via a selection of the i^{th} disjunct of a ground disjunctive clause C_r,

$$\begin{aligned} PITA(C_r, i) = PITA_h(h_i) \leftarrow\ & PITA_b(b_1), PITA_b(b_2), \\ & and(B_1, B_2, BB_2), \ldots, \\ & PITA_b(b_m), and(BB_{m-1}, B_m, BB_m), \\ & var(C_r, [], Var), \\ & equality(Var, i, B), and(BB_m, B, BDD). \end{aligned}$$

the transformed rule propagates conjunction as for non-disjunctive rules within the present iteration case (1b), and as with disjunctive facts (iteration case 1a) "creates" a new variable Var and a new BDD B indicating that the value of Var is i to represent the atomic choice $\{(C_r, \emptyset, i)\}$, and conjoins B to BB_m to produce the final BDD for the rule, which is an explanation for a.

In the case of $False_{\emptyset}$, an atom a removed from the greatest fixed point is such that at least one of one of its clauses does not have any positive body literal in the previous iteration of $False_{\emptyset}$. As a result a is not an unfounded atom with respect to the interpretation $\emptyset$. This condition is not affected by the PITA transformation.

2. *Inductive Case Outer Induction (WFM Operator)* This case consists of induction for $False_I$ and $False_I$, where I is the interpretation produced by the previous iteration of the WFM operator.

   (a) *Base Case Inner Induction ($True_I$ and $False_I$ Operators)*
   For $True_I$, in the case of non-disjunctive rules, atoms may be added to the interpretation by rules of the form

$$r = h \leftarrow b_1, \ldots, b_k, not(b_{k+1}), \ldots, not(b_m).$$

   where $b_1, \ldots, b_k$ are true in I, and $b_{k+1}, \ldots, b_m$ are false in I. Such rules are transformed into

$$\begin{aligned} PITA(C_r) = PITA_h(h) \leftarrow\ & PITA_b(b_1), \ldots, PITA_b(b_k), \\ & and(BB_{k-1}, B_k, BB_k), \\ & (PITA_b(b_{k+1}) \rightarrow not(BN_{k+1}, B_{k+1}); one(B_{k+1})), \\ & and(BB_k, B_{l_{k+1}}, BB_{k+1}), \ldots, \\ & and(BB_{m-1}, B_m, BDD). \end{aligned}$$

   This is similar to non-disjunctive rules seen in inductive case 1b with the addition of the negative literals that have been transformed into conditionals.

If a negative literal $PITA_b(b_{k+1})$ fails, conjoining the BDD 1 with BB_k in $and(BB_k, B_{k+1}, BB_{k+1})$ has no effect on the semantics of BB_{k+1} which is equal to BB_k. If $PITA_b(b_{k+1})$ succeeds, its complement is taken for the conjunction to produce a new BDD whose paths to 1 indicate the explanations[15].

In the case of disjunctive rules, an atom is added by a rule whose i^{th} disjunct is translated into

$$\begin{aligned} PITA(C_r, i) = PITA_h(h) \leftarrow\ & PITA_b(b_1), \ldots, PITA_b(b_k), \\ & and(BB_{k-1}, B_k, BB_k) \\ & (PITA_b(b_{k+1}) \rightarrow not(BN_{k+1}, B_{k+1}); one(B_{k+1})), \\ & and(BB_k, B_{k+1}, BB_{k+1}), \ldots, \\ & var(C_r, [], Var), equality(Var, i, B), \\ & and(BB_m, B, BDD). \end{aligned}$$

This is similar to the previous case, but with the addition of "adding" a new variable Var and a new BDD B indicating that the value of Var is i to represent the atomic choice $\{(C_r, \emptyset, i)\}$, just as for disjunctive rules for induction case (1b). As in that case B is conjoined to BB_m to produce the final BDD for the rule, which represents a finite set of explanations for a.

In the base case of $False_I$, those atoms removed from the greatest fixed point are those all of whose rules have at least one body literal that is false in I. This is unaffected by the PITA transformation.

(b) *Inductive Case (Outer)*

For $True_I$, from the perspective of the PITA transformation, the actions in this case are essentially the same as in inductive case 2a.

In the case of $False_I$, an atom a removed from the greatest fixed point is such that at least one of one of its clauses does not have either any positive body literal in the previous iteration of $False_I$ or a body literal that is false in I. As a result a is not an unfounded atom with respect to the interpretation I. This condition is not affected by the PITA transformation.

(Completeness ⇒) Sketch

The completeness proof of Theorem 1 maps interpretations over a LPAD T to $PITA(T)$. To accomplish this, note that the PITA transformation corresponds to the identity transformation for atoms whose predicate symbol is *and/3*, *or/3*, *not/2*, *one/1*, *zero/1*, *equality/3* and *var/3*.

For completeness we must prove that if κ is an explanation for a ground atom a then there is some instantiation $PITA_h(a)\theta \in WFM(PITA(ground(T)))$ of $PITA_h(a)$ such that such that κ is a path to a 1 leaf in $BDD(PITA_h(a)\theta)$.

Let $h \leftarrow body$ be a non-disjunctive clause in T. If $body$ is in $True_I(T')$ for interpetations I and T', then it is straightforward to show that $PITA(body)$ is also true in $True_I(T')$ – since the extra literals in $PITA(body)$ such as *and/3*, *one/1*, etc. have been added to the interpretations T' and I' (cf. the proof of Lemma 1).

The same argument holds for the body of a disjunctive clause

$$C = h_1 : \alpha_1 \vee \ldots \vee h_n : \alpha_n \leftarrow b_1, \ldots, b_m.$$

[15] Technically speaking, the Prolog conditional $(A \rightarrow B; C)$ must first be translated to $(A, B; \neg A, C)$ and this form translated into clauses without body disjuncts.

In addition, note that if (C, θ, i) is an atomic choice used to construct κ, then the PITA transformation ensures that a rule is generated for h_i, one instantiation of which corresponds to (C, θ, i).

Extending this to a full proof would involve placing these observations within the structure of a double induction like that used for Soundness. Note that because of Lemma 1 no transfinite induction would be needed for this induction. However the main ideas used in the cases of such an induction are simply those just mentioned combined with the analysis of the different cases of the PITA transformation shown for Soundness.

Definition 5. *A normal program P has the* bounded term size property *if there is a finite function $f : N \rightarrow N$ such that if a query term Q to P has size $size(Q)$ then no term used in the derivation of Q has size greater than $f(size(Q))$. An LPAD T has the bounded term size property if each $w \in W_T$ has the bounded term size property.*

Note that the query Q in Definition 5 need not be ground. From Definition 4 a program that is dynamically-finitary is such that for each *ground* query Q a bounded-term-size proof could be constructed (simply by computing the well-founded model until Q is seen to be true or false). Thus a bounded term-size program is dynamically-finitary, but the converse is false.

Theorem 2 *Let T be a range restricted, bounded term size, fixed-order dynamically stratified LPAD and a a ground atom. Let $\mathcal{E}$ be an SLG evaluation of $PITA_h(a)$ against $PITA(T)$, such that answer subsumption is declared on $PITA_h(a)$ using BDD-disjunction. Then $\mathcal{E}$ terminates with an answer ans for $PITA(a)$ and $BDD(ans)$ represents a covering set of explanations for a.*

Proof. (Sketch) The proof uses the forest-of-trees model [19] of SLG [3].

Because T is fixed-order dynamically stratified, queries to T can be evaluated using SLG without the DELAYING, SIMPLIFICATION or ANSWER COMPLETION oprations.. Instead, as [17] shows, only the SLG operations NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, ANSWER RETURN and NEGATIVE RETURN are needed. However, because $\mathcal{E}$ uses answer subsumption, an extension to the ANSWER RETURN operation must also be defined. The operation assumes answer subsumption on an upper semilattice L (implicitly on the last argument of the selected literal).

– SUBSUMING ANSWER RETURN: Let an SLG forest $\mathcal{F}_n$ contain a non-root node

$$N = Ans \leftarrow S, GoalList$$

whose selected literal S is positive and has been declared to use answer subsumption over a lattice L. Further, assume that S is ground in all arguments except its last, V_n, which is unbound. Let $\mathcal{A}$ be the set of all answers for S in $\mathcal{F}_n$, and $Join$ be the L-join of all the final arguments of all answers in $\mathcal{A}$. Assume that in $\mathcal{F}_n$, N does not have a child $S\{V_n/Join\}$. Then add $S\{V_n/Join\}$ as a child of N.

For the proof, the first item to note is that since T is range restricted, it is straightforward to show that $PITA(T)$ is also range restricted; that any subgoal $PITA_h(a)$ will be ground except for its last argument, which is free; and that all answers of $PITA_h(a)$

will be ground (cf. [12]). Accordingly the operation SUBSUMING ANSWER RETURN will be applicable to any subgoal with a non-empty set of answers.

Next, since T has the bounded term size property, any SLG evalation of a query Q to T will terminate [3], and it is straightforward to extend this result to SLG evaluations extended with answer subsumption.

It remains to show that answer ans for $PITA_h(a)$ is such that $BDD(ans)$ represents a covering set of explanations for a. The bounded term size property of T implies that there will only be a finite set of explanations (otherwise $f(size(Q))$ would be unbounded). Furthermore, each explanation in the set is finite, as follows from the fact that T is dynamically-finitary (as implied by the bounded term size property).

That $BDD(ans)$ contains a covering set of explanations can be shown by induction on the number of BDD operations. Consider an and operation in the body of a clause. For the inductive assumption, BB_{i-1} and B_i both represent finite covering set of explanations covering for $b_1, \ldots, b_{i-1}$ and b_i respectively. Let F_{i-1}, F_i', and F_i be the formulas expressed by BB_{i-1}, B_i, and BB_i respectively. These formulas can be represented in disjunctive normal form, in which every disjunct represents an explanation. F_i is obtained by multiplying F_{i-1} and F_i' so, by algebric manipulation, we can obtain a formula in disjunctive normal form in which every disjunct is the conjunction of two disjuncts, one from F_{i-1} and one from F_i'. Every disjunct is thus an explanation for F_i – the body prefix upto and including b_i. Moreover, every explanation for F_i is obtained by conjoining an explanations for F_{i-1} with an explanation for F_i', so it is represented by a disjunct of F_i.

In the case of an or operation bewteen two answers, the resulting BDD will represents the union of the set of explanations represented by the BDDs that are joined. Since this is repeated for all answers, the set of explanations represented by $BDD(ans)$ is covering for the query.

B Comparisons of Termination Properties

THIS IS STILL ROUGH, BUT ITS A START

Several classes of programs for which termination is studied make use of the (static) dependency graph of a program.

Definition 6. *The* dependency graph *of a program P is a labelled directed graph, denoted by $DG(P)$, whose vertices are the ground atoms of Ps language, and*

1. *there exists an edge labelled + (called a positive edge) from A to B iff for some rule $r \in Ground(P)$, $A \in head(r)$ and $B \in body(r)$;*
2. *there exists an edge labelled - (called a negative edge) from A to B iff for some rule $r \in Ground(P)$, $A \in head(r)$ and $\neg B in body(r)$*

An atom A depends on *an atom B if there is a path from A to B in the dependency graph of P.*

We begin by recalling the definitions of finitely recursive and finitary programs from [1] [16].

Definition 7. *[1] A program P is finitely recursive iff each ground atom A depends on finitely many ground atoms in $DG(P)$*

Definition 8. *[1] A program P is finitary if the following conditions hold:*

1. *P is finitely recursive.*
2. *There are finitely many odd-cyclic atoms in the dependency graph DG(P).*

It is straightforward to show that a finitely recursive program is dynamically-finitary. The following example shows that the converse is not the case.

Example 4. The program

p(f(0)). p(X)←q(X,Y),p(Y).

is not finitely recursive, but is dynamically-finitary (TLS: although an application of $False_I$ requires quantification over an infinite number of clauses).

We now define a superclass of bounded-term-size programs (Definition 5), based on constraining the types of allowed queries.

Definition 9. *Let Q be a ground atom. A normal program P has the* atomic bounded term size property *if there is a finite function $f : N \rightarrow N$ such that if a query Q to P has size $size(Q)$ then no term used in the derivation of Q has size greater than $f(size(Q))$.*

Example 5. The program P_{member}

member(X,[X—_]). member(X,[_—Y]):- member(X,Y).

[16] The definitions were originally given in terms of disjunctive programs; here we restate them in terms of normal programs.

has the atomic bounded term size property and is also dynamically-finitary (i.e. computation of $False_{\emptyset}$ would have an infinite fixed point, but computation of $\neg member(a, [b, c])$ would be determined within a finite number of iterations of that operator). Clearly P_{member} is not finitely recursive.

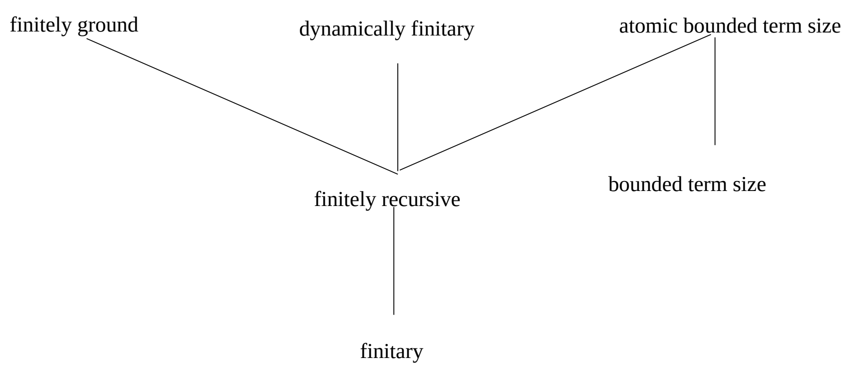
Next, [2] defines the concept of finitely ground programs. Such programs are amenable to grounding techniques that involve instantiation with respect to a given set of extensional database predicates, along with an iterative instantiation method that makes use of static dependency components. I CAN PUT THE DEFINITIONS IN AT SOME POINT.

Example 6. Consider the program

q(0). p(X)← q(X),p(s(X)).

Determining the falsity of e.g. *p(0)* using the fixed point operators of Section 5 requires detection of an unfounded set that is an infinite chain. Accordingly, this program is not dynamically-finitary, nor does it have the atomic bounded term size property, nor is it finitely recursive. However, an iterative instantiation method grounds the rule *p(X)← q(X),p(s(X)).* only with the binding $\{X = 0\}$ so that the program is finitely ground.

The putative hierarchy of programs is shown in Figure B NEEDS TO BE RECHECKED.



.