Artificial Intelligence 175 (2011) 890–913

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Loop-separable programs and their first-order definability

Yin Chen [a,*], Fangzhen Lin [b], Yan Zhang [c], Yi Zhou [c]

[a] Department of Computer Science, South China Normal University, Guangzhou, Guangdong, China
[b] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
[c] Intelligent Systems Lab, School of Computing and Mathematics, University of Western Sydney, Penrith South DC, NSW 1797, Australia

ARTICLE INFO

Article history:
Received 23 December 2009
Received in revised form 18 December 2010
Accepted 18 December 2010
Available online 22 December 2010

Keywords:
Answer set programming
First-order definability
Knowledge representation
Nonmonotonic reasoning

ABSTRACT

An answer set program with variables is first-order definable on finite structures if the set of its finite answer sets can be captured by a first-order sentence. Characterizing classes of programs that are first-order definable on finite structures is theoretically challenging and of practical relevance to answer set programming. In this paper, we identify a non-trivial class of answer set programs called loop-separable programs and show that they are first-order definable on finite structures.

1. Introduction

This work is about answer set programming (ASP), a constraint-based programming paradigm that has been found applications in a wide range of areas including bioinformatics [9,12,29] and the semantic web [11,27]. Currently in ASP applications, a program normally has two parts: a finite set of rules with variables, and a finite set of ground facts. The former represents general domain knowledge and the latter the specific instance of the problem that one wants to solve. Since current ASP solvers can only deal with rules without variables [14,20,22,28], the latter is used to ground the former into a set of propositional rules, and together they are given to an ASP solver.

Recently there has been work on extending answer set semantics to programs with variables [4,13,23,25], and to consider the possibility of constructing an ASP solver that can deal with rules with variables [4]. Against this backdrop, in this paper we consider the problem of first-order definability of answer set programs with variables. This is a problem because in general, the answer sets of a program with variables correspond to a second-order sentence [13,23] or an infinite set of first-order sentences [4].

The study on non-grounding based method for computing answer sets/stable models has been carried out by some researchers [10,16]. The motivation of developing this approach is to avoid large sets of facts after grounding a program containing variables. By introducing concepts such as constrained non-ground stables [10] and covers/anticovers [16], using this approach we can derive some kind of compact representations of the stable models of the original program, so that stable models may be partially pre-computed at compile-time.

Although both the approach mentioned above and the first-order definability of logic programs address non-grounding logic programs, the foundation of these two topics are actually quite different. In this paper, our study will be based on the first-order stable model semantics and identify a class of programs that is first-order definable on finite structures, while the non-ground approach only provided an alternative method to compute stable models of a propositional logic program.

* Corresponding author.
E-mail addresses: ychen@scnu.edu.cn (Y. Chen), flin@cse.ust.hk (F. Lin), yan@scm.uws.edu.au (Y. Zhang), yzhou@scm.uws.edu.au (Y. Zhou).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2010.12.001

While our work presented in this paper is the first in-deep study on the first-order definability of answer set programs with variables, we should mention that the related problem has been addressed in propositional case. In particular, Dung and Kanchanasut have shown that every propositional logic program Π can be transformed into a propositional theory T_Π such that the set of stable models of Π is exactly the set of models of T_Π [8]. More recently, Lin and Zhao proved a similar result by using loop-formulas [22].

Studying the first-order definability of answer set programs has both theoretical and practical values. Firstly, since the semantics of first-order answer set programs is defined based on second-order logic, it becomes an immediate issue to understand the expressive power of first-order answer set programs. Results of the first-order definability will provide partial answers to this issue and help us to achieve a better understanding on the difference between first-order answer set programs and classical first-order logic. Results in this aspect will provide an important theoretical foundation for first-order answer set programming.

Secondly, as evident from the work in Datalog and finite model theory, proving first-order definability results are usually highly challenging. Very often, new proof techniques have to be developed, which may also be useful for other problem solving. For instance, as it will be shown in this paper, in order to prove our first-order definability result, we extend the expansion tree concept in Datalog [3] to ASP and apply it to loop-separable programs. We believe that both the notion of loop-separable programs and the new expansion tree concept proposed in this paper may be useful for other related studies in first-order answer set programming.

Finally, knowing that a program is first-order definable is certainly helpful if one wants to construct an ASP solver for first-order answer set programs. It initiates the possibility of exploiting first-order inference tools, e.g. model generators and theorem provers, to reason on programs that are first-order reducible. Also, it can be helpful for SAT-based propositional ASP solvers. This is because current SAT-based ASP solvers compute loop formulas incrementally as needed. If we know that the given program can be captured by a first-order sentence, then it may be more effective to bypass loop formulas and just instantiate the first-order sentence on a given instance directly.

In this paper, we show that if a program is so-called *loop-separable*, then it is first-order definable on finite structures. Furthermore, it is decidable whether a program is loop-separable. As we shall see, the notion of loop-separable programs depends on a careful study of how rules interacts with first-order loops introduced in [4]. It also includes all first-order definable classes of programs that we knew of, like the class of program with finite set of complete loops.

The rest of the paper is organized as follows. Section 2 presents basic logic concepts and notions which will be used in our following study. Section 3 introduces the notion of first-order definability, and Section 4 defines a class of programs called loop-separable program. Section 5 contains the detailed proof that loop-separable programs are first-order definable. Section 6 considers some special subclasses of loop-separable programs and discusses some related work. Finally, Section 7 concludes this paper with some discussions.

2. First-order answer set programs with extensional databases

2.1. Preliminaries

We consider a second-order language with equality but without function symbols. A *vocabulary* consists of a finite set of *constant symbols* and a finite non-empty set of *relation symbols* including equality $=$. Given a vocabulary τ, we denote by $\mathcal{C}(\tau)$ the sets of constant symbols in τ, and by $\mathcal{P}(\tau)$ the set of relation symbols. The notions of *term*, *atom*, (first-order or second-order) *formula* and (first-order or second-order) *sentence* are defined as usual. An atom is called an *equality* atom if it is an atom of the form $t_1 = t_2$, and a *proper* atom otherwise. We use $Var(\mathcal{O})$ to denote the set of variables occurring in $\mathcal{O}$, which can be a term, atom, formula, sentence or other expressions. Given a vocabulary τ, the *unique name assumption* (or UNA for short) on τ, denoted by $\Sigma_{una}(\tau)$ (or Σ_{una} when τ is obvious from the context), is the conjunction of $c_i \neq c_j$ for any two different constant c_i, c_j in $\mathcal{C}(\tau)$.

Let P and Q be two relation symbols or variables of the same arity. $P < Q$ stands for the formula $\forall\bar{x}(P(\bar{x}) \supset Q(\bar{x})) \wedge \neg\forall\bar{x}(Q(\bar{x}) \supset P(\bar{x}))$. For the given tuples of relation symbols $\mathcal{P} = (P_1, \ldots, P_k)$ and $\mathcal{P}' = (P'_1, \ldots, P'_k)$, where all P_i and P'_i $(1 \leqslant i \leqslant k)$ have the same arity, we use $\mathcal{P} < \mathcal{P}'$ to denote the formula $\bigwedge_{i=1}^{k} \forall\bar{x}(P_i(\bar{x}) \supset P'_i(\bar{x})) \wedge \neg \bigwedge_{i=1}^{k} \forall\bar{x}(P'_i(\bar{x}) \supset P_i(\bar{x}))$.

A *finite structure* $\mathcal{A}$ of vocabulary τ is a tuple $(A, c_1^{\mathcal{A}}, \ldots, c_m^{\mathcal{A}}, R_1^{\mathcal{A}}, \ldots, R_n^{\mathcal{A}})$, where A is a finite set called the *domain* of $\mathcal{A}$, $c_i^{\mathcal{A}} \in A$ (the interpretation of constant c_i) $(1 \leqslant i \leqslant m)$, and $R_i^{\mathcal{A}}$ (the interpretation of a k-ary relation symbol R_i) $(1 \leqslant i \leqslant n)$, a k-ary relation on A. In the following, we use $Dom(\mathcal{A})$ to denote the domain of structure $\mathcal{A}$. Unless stated otherwise, the domains of all structures are assumed to be finite in this paper.

Given two tuples $\bar{s} = (s_1, \ldots, s_n)$ and $\bar{t} = (t_1, \ldots, t_n)$ of the same length, we use $\bar{s} = \bar{t}$ to denote the formula $\bigwedge_{i=1}^{n} s_i = t_i$, and $\bar{s} \neq \bar{t}$ the formula $\neg(\bar{s} = \bar{t})$. A *binding* is an expression of the form x/t, where x is a variable, and t a term, and a *substitution* is a set of bindings containing at most one binding for each variable. If φ is a first-order formula (term, tuple of terms, etc.), and θ a substitution, we denote by $\varphi\theta$ the result of replacing every free variable in φ according to θ.

Given a set of variables or relation variables $\mathcal{V}$ and a structure $\mathcal{A}$, an *assignment σ on $\mathcal{V}$ over $\mathcal{A}$* is function that assigns each variable in $\mathcal{V}$ to a domain element in $Dom(\mathcal{A})$ and each n-ary relation variable in $\mathcal{V}$ to an n-ary relation on $Dom(\mathcal{A})$. We write $(\mathcal{A}, \sigma) \models \varphi(\bar{x})$ to mean that φ is true in $\mathcal{A}$ under the assignment σ.

2.2. Syntax and semantics

We introduce the syntax and semantics of first-order answer set programs with extensional databases in this section. A *rule* is of the form:

$$a \leftarrow b_1, \ldots, b_k, \text{not}\, c_1, \ldots, \text{not}\, c_l, \tag{1}$$

where a is either a proper atom or $\bot$, and $b_1, \ldots, b_k, c_1, \ldots, c_l$ $(k, l \geqslant 0)$ are atoms. A rule is called a *constraint* if a is $\bot$. Given a rule r of form (1), we call a the *head* of r, denoted by $Head(r)$, $\{b_1, \ldots, b_k, \text{not}\, c_1, \ldots, \text{not}\, c_l\}$ the *body* of r, denoted by $Body(r)$, $\{b_1, \ldots, b_k\}$ the *positive body* of r, and $\{\text{not}\, c_1, \ldots, \text{not}\, c_l\}$ the *negative body* of r. We also use $Pos(r)$ and $Neg(r)$ to denote the set of atoms $\{b_1, \ldots, b_k\}$ and the set of atoms $\{c_1, \ldots, c_l\}$. A variable x in a rule r is called a *local variable* if it occurs in the body of r but not in the head of r.

A *first-order answer set program with extensional database* (or simply called *program*) is a finite set of rules. Given a program Π, we use $\tau(\Pi)$ to denote the vocabulary containing all the relation symbols and constants in Π. A relation symbol P in $\tau(\Pi)$ is called *intensional predicate* if it occurs in the head of some rules of Π, and *extensional predicates* otherwise. We use $\tau_{ext}(\Pi)$ to denote the vocabulary containing all extensional predicates and constants in Π, and $\tau_{int}(\Pi)$ the vocabulary containing all intensional predicates in Π. We also use $\mathcal{P}(\Pi)$, $\mathcal{P}_{int}(\Pi)$ and $\mathcal{P}_{ext}(\Pi)$ to denote the sets (tuples, if it is clear from the context) of all predicates, intensional predicates and extensional predicates in Π respectively. A proper atom $P(\bar{t})$ is extensional (intensional) if P is extensional (intensional).

Now we present the semantics of first-order answer set programs with extensional databases. For each rule r of form (1), we use $\widehat{r}$ to denote the sentence $\forall \bar{x}(\exists \bar{y}\widehat{Body_r} \supset a)$, where $\bar{y}$ is the tuple of all local variables of r and $\bar{x}$ the rest of variables in r, and $\widehat{Body_r}$ the formula $b_1 \wedge \cdots \wedge b_k \wedge \neg c_1 \wedge \cdots \wedge \neg c_l$. By $\widehat{\Pi}$, we denote the sentence $\bigwedge_{r\in\Pi} \widehat{r}$.

Let $\mathcal{P} = (P_1, \ldots, P_k)$ and $\mathcal{P}' = (P'_1, \ldots, P'_k)$ be two tuples of relation symbols or relation variables where P_i and P'_i $(1 \leqslant i \leqslant k)$ are of the same arity. Given a rule r of form (1), by $\widehat{r}[+\mathcal{P}/\mathcal{P}']$, we mean the formula that is obtained from $\widehat{r}$ by replacing each relation symbol in $\mathcal{P}$ occurring in the head and positive body of r by the corresponding relation symbol in $\mathcal{P}'$. We also use $\widehat{\Pi}[+\mathcal{P}/\mathcal{P}']$ to denote the formula $\bigwedge_{r\in\Pi} \widehat{r}[+\mathcal{P}/\mathcal{P}']$. For instance, if r is the rule $P(x) \leftarrow R(x), \text{not}\, Q(x)$, then $\widehat{r}[+\{P, Q\}/\{P', Q'\}]$ is the sentence $\forall x(R(x) \wedge \neg Q(x) \supset P'(x))$. Note that here we do not replace the relation symbol Q in the negative body of r.

Definition 1 *(Answer set).* Let Π be a program. A structure $\mathcal{A}$ of $\tau(\Pi)$ is an *answer set* of Π if and only if $\mathcal{A}$ is a model of

$$\widehat{\Pi} \wedge \neg\exists\mathcal{P}^*\big(\mathcal{P}^* < \mathcal{P}_{int}(\Pi) \wedge \widehat{\Pi}\big[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*\big]\big). \tag{2}$$

Example 1. We consider a program Π_1 consisting of the following rules:

$$T(x, y) \leftarrow E(x, y), \text{not}\, E(x, x), \text{not}\, E(y, y),$$

$$T(x, z) \leftarrow T(x, y), T(y, z),$$

where $\mathcal{P}_{ext}(\Pi_1) = \{E\}$ and $\mathcal{P}_{int}(\Pi_1) = \{T\}$. Let $\mathcal{A} = (A, E^{\mathcal{A}}, T^{\mathcal{A}})$ be a structure of $\tau(\Pi)$, where $E^{\mathcal{A}} = \{(a, a), (a, b), (b, c), (c, d)\}$ and $T^{\mathcal{A}} = \{(b, c), (c, d), (b, d)\}$. According to Definition 1, $\mathcal{A}$ is an answer set of Π. If we view E as a graph, then T computed by program Π_1 is the transitive closure of the induced subgraph of E on the set of nodes that do not have an edge going into themselves.

Note that in Definition 1, minimization only applies on intensional predicates while extensional predicates are viewed as the initial input of the program. This is different from the previous first-order answer set semantics such as Ferraris et al. [13] and Lin and Zhou's semantics [23]. There are both theoretical and practical advantages by separating a program vocabulary into intensional and extensional. Firstly, by separating intensional and extensional predicates in a program, the program itself may be viewed as a generic description of certain system or agent's behaviors, while the extensional predicates just provide various instantiations of the system or agent's initial inputs. Consequently, the class of programs that contain the same rules but with different extensional predicate inputs share many essential properties so that our study on these properties such as first-order definability and complexity may be simplified. Secondly, from a practical viewpoint, such separation will also simplify the underlying implementation for problem solving in various domains. This is the current practice in ASP anyway. For instance, we can easily write a generic program of computing Hamiltonian cycles for any finite graph without considering specific input graph – which will be represented by extensional predicate values.

In fact, the semantics presented above is nothing new by a simplification of the answer set (stable model) semantics recently presented by Ferraris et al. [13] and by Lin and Zhou [23]. The main differences are twofold. First, we distinguish between extensional and intensional predicates as discussed above. Second, here we only consider normal logic programs with constraints (i.e. programs without functions, disjunctions and nested expressions) rather than an arbitrary first-order sentence. Also, this definition goes back to the early work of Lin [21] by relating normal logic program under the stable model semantics and circumscription. As we will show next, under the context of finite structures, it is the same as the standard Gelfond–Lifschitz transformation semantics when the program is "grounded" on finite domains.

2.3. Relation to other answer set semantics

Given a program Π and a structure $\mathcal{A}$ of $\tau_{ext}(\Pi)$, we shall define *the instantiation of Π on $\mathcal{A}$* as a propositional program over the following propositional language $\mathcal{L}_{\mathcal{A}}$:

$$\mathcal{L}_{\mathcal{A}} = \{P(\bar{a}) \mid P \in \mathcal{P}_{int}(\Pi) \text{ and } \bar{a} \in Dom(\mathcal{A})^n\}.$$

We begin with one more notation. Let α be an atom and σ an assignment over $\mathcal{A}$. We denote by $\alpha[\sigma]$ the result of replacing every constant c in α by domain element $c^{\mathcal{A}}$ and every variable x in α by $\sigma(x)$.

Let $r \in \Pi$ be a rule of form (1). We define the instantiation of r on $\mathcal{A}$, written $r_{\mathcal{A}}$, to be the set of propositional rules obtained from

$$\mathcal{R} = \{a[\sigma] \leftarrow b_1[\sigma], \ldots, b_k[\sigma], \text{not } c_1[\sigma], \ldots, \text{not } c_l[\sigma] \mid \sigma \text{ is an assignment on } Var(r) \text{ over } \mathcal{A}\}$$

by the following transformations:

- if the body of a rule in $\mathcal{R}$ contains either $a = b$ for some distinct elements $a, b \in Dom(\mathcal{A})$ or $\text{not } a = a$ for some element $a \in Dom(\mathcal{A})$, then delete this rule;
- if the body of a rule in $\mathcal{R}$ contains either $P(\bar{a})$ for extensional predicate P and $\bar{a} \notin P^{\mathcal{A}}$ or $\text{not } P(\bar{a})$ for extensional predicate P and $\bar{a} \in P^{\mathcal{A}}$, then delete this rule;
- delete $a = a$ and $\text{not } a = b$ for all elements $a, b \in Dom(\mathcal{A})$ in the bodies of the remaining rules;
- delete $P(\bar{a})$ and $\text{not } P(\bar{a})$ in the bodies of the remaining rules, where P is an extensional predicate.

The instantiation of a program Π on $\mathcal{A}$, written $\Pi_{\mathcal{A}}$, is then the union of the instantiations of all the rules in Π on $\mathcal{A}$.

We also recall some definitions of answer set semantics for propositional program from [15]. Given a propositional language $\mathcal{L}$, a propositional program π is a finite set of propositional rules of the form:

$$pa \leftarrow pb_1, \ldots, pb_k, \text{not } pc_1, \ldots, \text{not } pc_l, \tag{3}$$

where pa is either $\bot$ or a propositional atom in $\mathcal{L}$, and $pb_1, \ldots, pb_k$, $pc_1, \ldots, pc_l$ $(k, l \geqslant 0)$ are propositional atoms. A propositional program π is called *positive* if $l = 0$ for all rules of form (3) in π. Given a set of propositional atoms $M \subseteq \mathcal{L}$ and a propositional program π, we use $GL_M(\pi)$ to denote the propositional program obtained from π by the following transformations:

- if a rule of form (3) is in π and $pc_i \in M$ for some i, $1 \leqslant i \leqslant l$, then delete this rule;
- delete $\text{not } pc_i$, $1 \leqslant i \leqslant l$, in the bodies of the remaining rules.

A set of propositional atoms $M \subseteq \mathcal{L}$ is an answer set of a propositional program π if it is the minimal set of propositional atoms that *satisfies* every rule in $GL_M(\pi)$, where M satisfies a rule of form (3) if

- either pa is $\bot$, and $\{pb_1, \ldots, pb_k\} \nsubseteq M$ or $\{pc_1, \ldots, pc_l\} \cap M \neq \emptyset$,
- or pa is a propositional atom, and $pa \in M$ whenever $\{pb_1, \ldots, pb_k\} \subseteq M$ and $\{pc_1, \ldots, pc_l\} \cap M = \emptyset$.

Proposition 1. *Given a program Π and a finite structure $\mathcal{A}$ of $\tau_{ext}(\Pi)$, let $\mathcal{A}'$ be a structure of $\tau(\Pi)$ such that $Dom(\mathcal{A}') = Dom(\mathcal{A})$, $c^{\mathcal{A}'} = c^{\mathcal{A}}$ for every constant c, and $\bar{a} \in P^{\mathcal{A}'}$ if and only if $\bar{a} \in P^{\mathcal{A}}$ for every extensional predicate P. $\mathcal{A}'$ is an answer set of Π if and only if $M_{\mathcal{A}'}$ is an answer set of $\Pi_{\mathcal{A}}$, where $M_{\mathcal{A}'} = \{P(\bar{a}) \mid P \in \mathcal{P}_{int}(\Pi) \text{ and } \bar{a} \in P^{\mathcal{A}'}\}$.*

Proof. Assume that $\mathcal{P}_{int}(\Pi) = \{P_1, \ldots, P_n\}$, and let $\mathcal{P}^* = \{P_1^*, \ldots, P_n^*\}$ be a set of relation variables such that P_i and P_i^* $(1 \leqslant i \leqslant n)$ are of the same arity.

"⇒": $\mathcal{A}'$ is an answer set of Π. By Definition 1, $\mathcal{A}'$ is a model of (2), so it is a model of $\widehat{\Pi}$. By the definition of $\Pi_{\mathcal{A}}$ and $M_{\mathcal{A}'}$, we can see that $M_{\mathcal{A}'}$ satisfies every rules in $\Pi_{\mathcal{A}}$ and $GL_{M_{\mathcal{A}'}}(\Pi_{\mathcal{A}})$. We will show that $M_{\mathcal{A}'}$ is the minimal set which satisfies every rules in $GL_{M_{\mathcal{A}'}}(\Pi_{\mathcal{A}})$.

Otherwise, there is a set $M'' \subset M_{\mathcal{A}'}$ which also satisfies every rules in $GL_{M_{\mathcal{A}'}}(\Pi_{\mathcal{A}})$. We can see that M'' also satisfies every rules in $\Pi(\mathcal{A})$. Let σ be an assignment on $\mathcal{P}^*$ such that $\bar{a} \in \sigma(P_i^*)$ if and only if $P_i(\bar{a}) \in M''$, for every intensional predicate P_i $(1 \leqslant i \leqslant n)$. We will show that $(\mathcal{A}', \sigma) \models \widehat{\Pi}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$.

Let $r \in \Pi$ be a rule of form (1), and σ' an assignment on $Var(r)$. Consider the propositional rule

$$r' : a[\sigma'] \leftarrow b_1[\sigma'], \ldots, b_k[\sigma'], \text{not } c_1[\sigma'], \ldots, \text{not } c_l[\sigma'],$$

we can see that either

- there is $a = b$ for some distinct elements $a, b \in Dom(\mathcal{A})$ or $\text{not } a = a$ for some element $a \in Dom(\mathcal{A})$ in r', or
- there is $P(\bar{a})$ for extensional predicate P and $\bar{a} \notin P^{\mathcal{A}}$ or $\text{not } P(\bar{a})$ for extensional predicate P and $\bar{a} \in P^{\mathcal{A}}$,

or

- M'' satisfies propositional rule r'', where r'' is obtained by removing all the equality atoms and extensional atoms in the body of r'.

By considering both cases, we have $(\mathcal{A}', \sigma) \models \widehat{r}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$, and then $(\mathcal{A}', \sigma) \models \widehat{\Pi}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$. So, we have $(\mathcal{A}', \sigma) \models (\mathcal{P}^* < \mathcal{P}_{int}(\Pi)) \wedge \widehat{r}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$ by noticing that $M'' \subset M_{\mathcal{A}'}$, which is a contradiction to the fact that $\mathcal{A}'$ is a model of (2).

"⇐": $M_{\mathcal{A}'}$ is an answer set of $\Pi_{\mathcal{A}}$. By the definitions of $\Pi_{\mathcal{A}}$ and $M_{\mathcal{A}'}$, we can see that $\mathcal{A}'$ is a model of $\widehat{\Pi}$. It is sufficient to show that there does not exist an assignment σ on $\mathcal{P}^*$ such that $(\mathcal{A}', \sigma) \models (\mathcal{P}^* < \mathcal{P}_{int}(\Pi)) \wedge \widehat{r}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$.

Otherwise, let σ be an assignment on $\mathcal{P}^*$ such that $(\mathcal{A}', \sigma) \models (\mathcal{P}^* < \mathcal{P}_{int}(\Pi)) \wedge \widehat{r}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$. Let M'' be a subset of $\mathcal{L}_{\mathcal{A}}$ such that $P_i(\bar{a}) \in M''$ if and only if $\bar{a} \in \sigma(P_i^*)$, for every intensional predicate P_i $(1 \leqslant i \leqslant n)$. We will show next that M'' satisfies every rules in $\Pi_{\mathcal{A}}$.

Let $r'' \in \Pi_{\mathcal{A}}$ be a propositional rule obtained from the propositional rule

$$r' : a[\sigma'] \leftarrow b_1[\sigma'], \ldots, b_k[\sigma'], \mathsf{not}\, c_1[\sigma'], \ldots, \mathsf{not}\, c_l[\sigma'],$$

where there is a rule r of form (1) in Π, and σ' is an assignment on $Var(r)$. By the definition of $\Pi_{\mathcal{A}}$,

- each equality atom in the body of r' is of the form $a = a$ or $\mathsf{not}\, a = b$, where a and b are distinct elements in $Dom(\mathcal{A})$,
- if $P(\bar{a}) \in Pos(r')$, then $\bar{a} \in P^{\mathcal{A}}$, for extensional predicate P,
- if $P(\bar{a}) \in Neg(r')$, then $\bar{a} \notin P^{\mathcal{A}}$, for extensional predicate P.

We already have $(\mathcal{A}', \sigma) \models \widehat{r}[+\mathcal{P}_{int}(\Pi)/\mathcal{P}^*]$. So, by the definition of M'', we can see that M'' satisfies rule r''. Furthermore, by $(\mathcal{A}', \sigma) \models \mathcal{P}^* < \mathcal{P}_{int}(\Pi)$, we can see that $M'' \subset M_{\mathcal{A}'}$, and M'' also satisfies every rules in $GL_{M_{\mathcal{A}'}}(\Pi_{\mathcal{A}})$ by noticing that $M_{\mathcal{A}'}$ satisfies every rules in $\Pi_{\mathcal{A}}$. This is a contradiction to the fact that $M_{\mathcal{A}'}$ an answer set of $\Pi_{\mathcal{A}}$. □

Proposition 1 indicates that Definition 1 coincides with the standard Gelfond–Lifschitz semantics but lifted to an arbitrary finite structures rather than only considering the Herbrand structure [15]. As a consequence, Definition 1 also coincides with Ferraris et al.'s recent semantics [13] restricted to normal logic programs with constraints on finite structures.

Corollary 2. *Let Π be a program such that all predicates in Π are intensional. A structure $\mathcal{A}$ is a stable model of $\widehat{\Pi}$ under Ferraris et al.'s definition* [13] *iff it is an answer set of Π under Definition* 1.

Another issue is that we distinguish between intensional predicates and extensional predicates in this paper. In fact, the main purpose is conceptual but not technical. Also followed from Proposition 1, the following two properties show that programs with and without extensional predicates can be simply transformed from one another. More precisely, from programs without extensional predicates to those with, one can add "identity rules" of the form

$$P(\bar{x}) \leftarrow P(\bar{x})$$

for every predicate P in Π. For the other way around, one can add "choice rules" of the form

$$P'(\bar{x}) \leftarrow \mathsf{not}\, P(\bar{x}),$$
$$P(\bar{x}) \leftarrow \mathsf{not}\, P'(\bar{x})$$

for every extensional predicate P, where P' is a new predicate that has the same arity as P.

Corollary 3. *Let Π be a program. A structure $\mathcal{A}$ is a stable model of $\widehat{\Pi}$ under Ferraris et al.'s definition* [13] *iff it is an answer set of $\Pi \cup ID(\Pi)$ under Definition* 1*, where $ID(\Pi)$ is the set of all identity rules for all predicates in Π.*

Corollary 4. *Let Π be a program. A structure $\mathcal{A}$ is an answer set of Π under Definition* 1 *iff $\mathcal{A}'$ is a stable model of $\widehat{\Pi} \wedge \widehat{Choice(\Pi)}$ under Ferraris et al.'s definition* [13]*, where $\mathcal{A}'$ is the conservative extension of $\mathcal{A}$ under $\bigwedge_{P \in \tau(\Pi)} \forall \bar{x}(P(\bar{x}) \leftrightarrow \neg P'(\bar{x}))$, and $Choice(\Pi)$ is the set of all choice rules for all extensional predicates in Π.*

Recently, Pelov, Denecker and Bruynooghe also introduced an alternative first-order extension of logic programs with aggregates under stable semantics [26]. In their formalism, a program is a (possibly infinite) set of aggregate rules of the form $A \leftarrow \varphi$, where A is an atom and φ is a first-order formula potentially including aggregate expressions. By defining a three-value immediate consequence operator of an aggregate program, they defined the extended stable semantics for such aggregate programs. Nevertheless, by ignoring the issue of aggregates, it is not difficult to observe that their fixpoint based extended stable semantics actually coincides with Gelfond–Lifschitz's original stable model semantics on grounded programs [15].

2.4. Loops and loop formulas

For finite domains, the answer set semantics of a first-order logic program can also be captured by loop formulas [4]. In the following, we review some results about loops and loop formulas as they will be used in defining a class of first-order definable programs.[1]

Given a program Π, the *positive dependency graph* of Π, denoted by G_Π, is the infinite graph (V, E), where V is the set of atoms of $\tau_{int}(\Pi)$, and (α, β) is an edge in E if there is a rule $r \in \Pi$ and a substitution θ, such that $\alpha = Head(r\theta)$ and $\beta \in Pos(r\theta)$. A finite non-empty subset L of V is said to be a *loop* of Π if there exists a cycle in G_Π that goes through only and all the nodes in L. In particular, for each atom $\alpha \in V$, we treat $\{\alpha\}$ as a special loop, in which there is a singleton. A rule r is said to be *involved* in a loop if there is a loop L and two atoms α and β in L such that $\alpha = Head(r)$ and $\beta \in Pos(r)$.

Let r be a rule of form (1), and suppose that a is $P(t_1, \ldots, t_n)$ for some predicate P and tuple $(t_1, \ldots, t_n)$ of terms. If $\bar{x} = (x_1, \ldots, x_n)$ is a tuple of variables not in r, then the *normal form* of r on $\bar{x}$ is the following rule:

$$P(x_1, \ldots, x_n) \leftarrow x_1 = t_1, \ldots, x_n = t_n, b_1, \ldots, b_k, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_l.$$

Given a loop L of a program Π and an atom $P(\bar{t})$ in L, the *external support formula* of $P(\bar{t})$ for Π with respect to L, denoted by $ES(P(\bar{t}), L, \Pi)$, is the following formula:

$$\bigvee_{1 \leqslant i \leqslant k} \exists \overline{y_i} \Big(\widehat{Body_{r_i\theta}} \wedge \bigwedge_{Q(\bar{t}') \in L,\ Q(\bar{t}) \in Pos(r_i\theta)} \bar{t} \neq \bar{t}' \Big), \tag{4}$$

where

- $r_1, \ldots, r_k$ are the normal forms on $\bar{x}$ of rules in Π whose head mention the predicate P;
- $\bar{x}$ is a tuple of variables that are not in Π, and if $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{x} = (x_1, \ldots, x_n)$, then $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ (so that $\bar{x}\theta = \bar{t}$);
- $\overline{y_i}$ $(1 \leqslant i \leqslant k)$ is the tuple of local variables of r_i.

The *loop formula* of L in Π, denoted by $LF(L, \Pi)$, is the universal closure of

$$\bigvee_{\alpha \in L} \alpha \supset \bigvee_{\alpha \in L} ES(\alpha, L, \Pi). \tag{5}$$

We use $LF(\Pi)$ to denote the set of all loop formulas in Π.

Theorem 1. *(See [4].) Let Π be a program[2] and $\mathcal{A}$ a finite structure of $\tau(\Pi)$, where $\mathcal{A}$ is a model of Σ_{una}. $\mathcal{A}$ is an answer set of Π if and only if $\mathcal{A}$ is a model of $\{\widehat{\Pi}\} \cup LF(\Pi)$.*

In general case, a program may have infinite many loops, and thus $LF(\Pi)$ is an infinite set of sentences (see programs Π_4 and Π_5 illustrated in next section). However, there are programs whose loop formulas can be captured by a finite set of formulas.

A *complete set of loops* S of a program Π is a set of loops such that for every loop L of Π, there is a loop $L' \in S$ and a substitution θ such that $L = L'\theta$. If a program Π has a finite complete set of loops S, then a finite structure $\mathcal{A}$ is an answer set of Π if and only if $\mathcal{A}$ is a model of the sentence $\widehat{\Pi} \wedge \bigwedge_{L \in S} LF(L, \Pi)$.

Example 2. Consider the following program Π_2:

$$r_1: \quad P(x) \leftarrow Q(x), R_1(x),$$
$$r_2: \quad Q(x) \leftarrow P(x), R_2(x),$$
$$r_3: \quad P(x) \leftarrow R_3(x),$$
$$r_4: \quad Q(x) \leftarrow R_4(x).$$

One of the complete set of loops of Π_2 is $\{\{P(y)\}, \{Q(y)\}, \{P(y), Q(y)\}\}$, and thus a structure $\mathcal{A}$ is an answer set of Π_2 if and only if it is a model of the conjunction of $\widehat{\Pi_2}$ and the following loop formulas:

$$\forall y \quad \big(P(y) \supset \big(Q(y) \wedge R_1(y)\big) \vee R_3(y)\big),$$
$$\forall y \quad \big(Q(y) \supset \big(P(y) \wedge R_2(y)\big) \vee R_4(y)\big),$$
$$\forall y \quad \big(\big(P(y) \vee Q(y)\big) \supset R_3(y) \vee R_4(y)\big).$$

[1] Some notions are a little bit different from [4] to fit our context.

[2] The theorem in [4] consider only programs without constraint, while it is straightforward to extend it to arbitrary programs.

3. First-order definable programs

Now we present a formal definition of first-order definability for answer set programs.

Definition 2. A program Π is called *first-order definable* on finite structures (under answer set semantics) if there is a first-order sentence ψ on vocabulary $\tau(\Pi)$ such that for every finite structure $\mathcal{A}$ of $\tau(\Pi)$, $\mathcal{A}$ is a model of ψ if and only if $\mathcal{A}$ is an answer set of Π. In this case, we say that Π *is defined by* ψ.

This notion of first-order definability for answer set programs is closely related to the well-studied definability problem of datalog queries in deductive database [1,6]. A *datalog rule* is a rule without negation, and a *datalog program* is a finite set of datalog rules, i.e., a program without negation. For any datalog program Π, it can be shown that given any database (structure) of extensional predicates, there is a unique answer set of Π based on the database. This unique answer set is what is computed by the datalog program from the given database of the extensional predicates.

A *datalog query* (Q, Π) $(Q \in \mathcal{P}_{int}(\Pi))$ represents the intended value of Q on a given extensional database $\mathcal{A}$ of Π, denoted as $Q(\mathcal{A})$. (Q, Π) is *explicitly first-order definable* if and only if there exists a first-order formula $\phi(\bar{x})$ on $\tau_{ext}(\Pi)$ such that for every extensional database $\mathcal{A}$ of Π, $Q(\mathcal{A})$ is the same as the relation represented by $\phi(\bar{x})$ under $\mathcal{A}$. (Q, Π) is *implicitly first-order definable* if there is a first-order sentence ϕ on $\tau_{ext}(\Pi) \cup \{Q\}$ such that for every extensional database $\mathcal{A}$ of Π and a relation R on $\mathcal{A}$, ϕ is satisfied in the extension of $\mathcal{A}$ with R if and only if $R = Q(\mathcal{A})$ [17]. It has been shown that explicit definability implies implicit definability, but the converse is not true in general on finite structures [1].

Our definition of first-order definability is similar to implicit first-order definability in Datalog. One main difference is that the first-order definability issue considered in ASP is about a program but not a query. Another difference is that in our definition we require a first-order sentence to capture all answer sets and thus all intensional predicates, not just the single intensional predicate mentioned in the query.

Example 3. Consider the following program Π_3:

$$P(x) \leftarrow Q(x), \text{not } R_1(x),$$
$$Q(x) \leftarrow R_2(x).$$

Π_3 has two intensional predicates: P and Q. According to Definition 2, Π_3 can be defined by the sentence

$$\forall x\big(P(x) \equiv \big(Q(x) \wedge \neg R_1(x)\big)\big) \wedge \forall x\big(Q(x) \equiv R_2(x)\big).$$

From our remarks at the end of last section, we see that if a program has a finite complete set of loops, then it is first-order definable. However, the converse is not true in general. As the following examples show.

Example 4. Consider the program Π_4:

$$r_1: \quad P(x) \leftarrow R(x), \tag{6}$$
$$r_2: \quad P(x) \leftarrow P(y), S(x,u), \text{not } T(y,v). \tag{7}$$

For this program, P is the only intensional predicate. Clearly, Π_4 does not have a finite complete set of loops: for each $n > 0$, $\{P(x_1), \ldots, P(x_n)\}$ is a loop.

Now consider how $P(x_1)$ can be derived. There are two rules for it:

$$P(x_1) \leftarrow R(x_1), \tag{8}$$
$$P(x_1) \leftarrow P(x_2), S(x_1,u_1), \text{not } T(x_2,v_1). \tag{9}$$

The first one is a *base rule*, where a rule is called a base rule if all predicates in the body of the rule are extensional. The second one has a recursive call to $P(x_2)$. Expanding the recursive call in the second rule produces two new rules for $P(x_1)$:

$$P(x_1) \leftarrow R(x_2), S(x_1,u_1), \text{not } T(x_2,v_1), \tag{10}$$
$$P(x_1) \leftarrow P(x_3), S(x_2,u_2), \text{not } T(x_3,v_2), S(x_1,u_1), \text{not } T(x_2,v_1). \tag{11}$$

Again, the first one is a base rule, and the second one has a recursive call to $P(x_3)$ which can be further expanded to produce more rules about $P(x_1)$. However, these new rules are really redundant. For instance, expanding $P(x_3)$ in (11) using rule (6) produces the following rule:

$$P(x_1) \leftarrow R(x_3), S(x_2,u_2), \text{not } T(x_3,v_2), S(x_1,u_1), \text{not } T(x_2,v_1). \tag{12}$$

But this rule is subsumed by (10). In fact, one can show that this program is equivalent to the program Π_4' which contains two rules (8) and (10). Π_4' has no loops and is defined by the conjunction of $\widehat{\Pi_4'}$ and

$$\forall x\big(P(x) \supset R(x) \vee \exists yuv\big(R(y) \wedge S(x,u) \wedge \neg T(y,v)\big)\big).$$

This is an example where recursive expansion terminates in one step. As we shall see, one reason is that the body of r_2 can be separated into two parts, $B_h = \{S(x,u)\}$ and $B_b = \{P(y), \text{not}\, T(y,v)\}$, such that B_h and the head of rule r_2 do not share any variables with B_b.

Example 5. Consider another program Π_5:

r_1: $P(x) \leftarrow R_1(x)$,

r_2: $Q(x,y) \leftarrow R_2(x,y)$,

r_3: $P(x) \leftarrow Q(x,y), \text{not}\, T(y,v)$,

r_4: $Q(x,y) \leftarrow P(x), S(x,u)$.

For this program, the intensional predicates are P and Q. We can see that Π_5 does not have a finite complete set of loops either: for each $n > 0$, $\{P(x), Q(x,y_1), \ldots, Q(x,y_n)\}$ is a loop. Notice here that in every loop, all atoms share a common variable x.

Just like Example 4, the recursive rules r_3 and r_4 only need to be expanded a finite number of times. For r_3, expanding $Q(x,y)$ in its body once produces one base rule and one recursive rule of the form $P(x) \leftarrow P(x), \ldots$ which can be discarded. For r_4, expanding $P(x)$ in its body once produces a base rule and the following recursive rule:

$$Q(x,y) \leftarrow Q(x,y_1), S(x,u), \text{not}\, T(y_1,v). \tag{13}$$

Expanding $Q(x,y_1)$ in (13) using r_2 produces a base rule:

$$Q(x,y) \leftarrow R_2(x,y_1), S(x,u), \text{not}\, T(y_1,v).$$

Expanding $Q(x,y_1)$ in (13) using r_4 produces a recursive rule:

$$Q(x,y) \leftarrow P(x), S(x,u_1), S(x,u), \text{not}\, T(y_1,v). \tag{14}$$

(14) is subsumed by r_4. It is redundant, and do not need to be expanded.

On further examination, we found that what makes this program first-order definable is that in every loop that "uses" the recursive rule r_3 (and similarly r_4), the body of the rule can be separated into two parts, $B_h = \emptyset$ and $B_b = \{Q(x,y), \text{not}\, T(y,v)\}$, such that the variables in both $B_h \cup \{Head(r_3)\}$ and B_b are the same as the variables in all the atoms in this loop.

4. Loop-separable programs

We have seen two examples above that do not have finite complete sets of loops but are nonetheless first-order definable. They are what we will call loop-separable programs. Our main result of this paper is in showing that all such programs are first-order definable. To define these programs, we need to first introduce some additional notions about loops.

4.1. Loop patterns

The two examples above suggest that when the variables in the rules involved in the loops of a program have some "patterns", its loop formulas are then "well-behaved" in the sense that they do not grow infinitely long. However, loops are sets of atoms, and the variables in a loop may not be the same as the variables occurring in the rules involved in the loop. This motivates our following definition of *derivation paths*, which extends the notion of path in the dependency graph of a program.

Definition 3. A *derivation path* S of a program Π is a finite sequence of pairs of an atom and a rule:

$$(\alpha_1, \rho_1), \ldots, (\alpha_n, \rho_n) \tag{15}$$

where

- for $1 \leqslant i \leqslant n$, there is a rule $r_i \in \Pi$ and a substitution $(\overline{x_i}/\overline{y_i})$ such that $\rho_i = r_i(\overline{x_i}/\overline{y_i})$, where $\overline{x_i}$ is the tuple of all the variables in r_i and for any local variable x_{ij} of r_i, y_{ij} is a new variable not in (α_k, ρ_k) $(1 \leqslant k < i)$;
- for $1 \leqslant i \leqslant n$, $\alpha_i = Head(\rho_i)$;
- for $1 \leqslant i < n$, $\alpha_{i+1} \in Pos(\rho_i)$.

If there is a path starting from an atom and ending at the same atom, then there is a loop. For derivation path, we also need to know when there is a cycle. In the following, we will define a relation between two pairs of an atom and a rule.

Let $\bar{t}_1 = (t_{11}, \ldots, t_{1n})$ and $\bar{t}_2 = (t_{21}, \ldots, t_{2n})$ be two tuples of terms of the same length. We define $\bar{t}_1 \sim \bar{t}_2$ if

- for every i $(1 \leqslant i \leqslant n)$ and constant c, $t_{1i} = c$ if and only if $t_{2i} = c$;
- for every i $(1 \leqslant i \leqslant n)$ and variable x, if $x \in \bar{t}_1$ and $x \in \bar{t}_2$, then $t_{1i} = x$ if and only if $t_{2i} = x$;
- for every i $(1 \leqslant i \leqslant n)$ and variable x, if $x \in \bar{t}_1$ and $x \notin \bar{t}_2$, then there is a variable y such that $y \in \bar{t}_2$ and $y \notin \bar{t}_1$, and $t_{1i} = x$ if and only if $t_{2i} = y$;
- for every i $(1 \leqslant i \leqslant n)$ and variable x, if $x \in \bar{t}_2$ and $x \notin \bar{t}_1$, then there is a variable y such that $y \in \bar{t}_1$ and $y \notin \bar{t}_2$, and $t_{2i} = x$ if and only if $t_{1i} = y$.

We also use $\bar{t}_1 \nsim \bar{t}_2$ if it is not the case $\bar{t}_1 \sim \bar{t}_2$. For example, we have $(x, y, x, z) \sim (x, u, x, w)$ and $(x, y, z) \nsim (y, x, u)$.

Intuitively, $\bar{t}_1 \sim \bar{t}_2$ represents a special mutual substitution relation between $\bar{t}_1$ and $\bar{t}_2$, where variables and their corresponding positions occurring in $\bar{t}_1$ and $\bar{t}_2$ cannot be mixed. Precisely, these properties are presented in the following proposition.

Proposition 5. *Let $\bar{t}_1$ and $\bar{t}_2$ be two tuples of terms of length n, and $\bar{t}_1 \sim \bar{t}_2$. Then, there is a substitution $\bar{x}/\bar{y} = \{x_1/y_1, \ldots, x_k/y_k\}$, where x_i and y_i $(1 \leqslant i \leqslant k)$ are variables, and*

- *for any $1 \leqslant i < j \leqslant k$, x_i and x_j are different variables*;
- *for any $1 \leqslant i < j \leqslant k$, y_i and y_j are different variables*;
- *$\{x_1, \ldots, x_k\} \cap Var(\bar{t}_2) = \emptyset$ and $\{y_1, \ldots, y_k\} \cap Var(\bar{t}_1) = \emptyset$*;
- *$\bar{t}_1(\bar{x}/\bar{y}) = \bar{t}_2$ and $\bar{t}_2(\bar{y}/\bar{x}) = \bar{t}_1$.*

Proof. Let $x_1, \ldots, x_k$ be the variables only in $\bar{t}_1$ and $y_1, \ldots, y_k$ the variables only in $\bar{t}_2$, such that $t_{1i} = x_j$ if and only if $t_{2i} = y_j$, $1 \leqslant i, j \leqslant k$. □

In the following, we call the substitution $(\bar{x}/\bar{y})$, as shown in Proposition 5, the *witness* of $\bar{t}_1 \sim \bar{t}_2$. For example, $\{y/u, z/w\}$ is the witness of $(x, y, x, z) \sim (x, u, x, w)$.

We extend relation $\sim$ to atoms, rules and pairs of an atom and a rule. For two atoms α_1 and α_2, where $\alpha_1 = P(\bar{t}_1)$ and $\alpha_2 = P(\bar{t}_2)$, we use $\alpha_1 \sim \alpha_2$ if $\bar{t}_1 \sim \bar{t}_2$. For two rules ρ_1 and ρ_2, where $\rho_1 = r(\bar{x}/\overline{y_1})$, $\rho_2 = r(\bar{x}/\overline{y_2})$ and $\bar{x}$ is the tuple of all variables in r, we use $\rho_1 \sim \rho_2$ if $\overline{y_1} \sim \overline{y_2}$. For two pairs (α_1, ρ_1) and (α_2, ρ_2), where α_1, α_2 are atoms, and ρ_1, ρ_2 are rules, we use $(\alpha_1, \rho_1) \sim (\alpha_2, \rho_2)$ if $\alpha_1 \sim \alpha_2$ and $\rho_1 \sim \rho_2$. We also use $\mathcal{O}_1 \nsim \mathcal{O}_2$, if it is not the case $\mathcal{O}_1 \sim \mathcal{O}_2$, where $\mathcal{O}_1$ and $\mathcal{O}_2$ can be two atoms, rules or pairs of an atom and a rule.

Now we can give the definition of *loop pattern*.

Definition 4. A derivation path of form (15) is called a *loop pattern* if $n > 1$, $\rho_1 \sim \rho_n$, and $\rho_i \nsim \rho_j$ for any other i, j $(1 \leqslant i, j \leqslant n)$.

Example 6. We continue with the program Π_4 in Example 4 and Π_5 in Example 5.

For program Π_4, its loop patterns are of the form:

$$\begin{aligned} lp_1: \quad & \big(P(x_1), r_2(x/x_1, y/x_2, u/u_1, v/v_1)\big), \\ & \big(P(x_2), r_2(x/x_2, y/x_3, u/u_2, v/v_2)\big), \\ & \big(P(x_3), r_2(x/x_3, y/x_4, u/u_3, v/v_3)\big). \end{aligned} \tag{16}$$

Note that

$$r_2(x/x_1, y/x_2, u/u_1, v/v_1) \sim r_2(x/x_3, y/x_4, u/u_3, v/v_3),$$

but

$$r_2(x/x_1, y/x_2, u/u_1, v/v_1) \nsim r_2(x/x_2, y/x_3, u/u_2, v/v_2)$$

and

$$r_2(x/x_2, y/x_3, u/u_2, v/v_2) \nsim r_2(x/x_3, y/x_4, u/u_3, v/v_3).$$

For program Π_5, there are two forms of loop patterns:

$$lp_1: \quad \big(P(x_1), r_3(x/x_1, y/x_2, v/v_1)\big),$$
$$\big(Q(x_1, x_2), r_4(x/x_1, y/x_2, u/u_1)\big),$$
$$\big(P(x_1), r_3(x/x_1, y/x_3, v/v_2)\big), \tag{17}$$
$$lp_2: \quad \big(Q(x_1, x_2), r_4(x/x_1, y/x_2, u/u_1)\big),$$
$$\big(P(x_1), r_3(x/x_1, y/x_3, v/v_1)\big),$$
$$\big(Q(x_1, x_3), r_4(x/x_1, y/x_3, u/u_2)\big). \tag{18}$$

The following proposition shows the relationship between loop patterns and loops.

Proposition 6. *If S is a loop pattern of form* (15) *of a program Π, then $\{\alpha_1(\bar{y}/\bar{x}), \ldots, \alpha_n(\bar{y}/\bar{x})\}$ is a loop of Π, where $(\bar{x}/\bar{y})$ is the witness of $\alpha_1 \sim \alpha_n$.*

Proof. There is a path from $\alpha_1(\bar{y}/\bar{x})$ to $\alpha_n(\bar{y}/\bar{x})$, and $\alpha_1(\bar{y}/\bar{x}) = \alpha_n(\bar{y}/\bar{x})$. □

There are programs that do not have a finite complete set of loops, while the following proposition shows that every program has a finite complete set of loop patterns.

Proposition 7. *For every program Π, there is a finite set of loop patterns $\mathcal{S}_l$ such that for every loop pattern S of Π, there is a loop pattern $S' \in \mathcal{S}_l$ and $S = S'\theta$, where $\theta = (\bar{x}/\bar{y})$ is a substitution, and all variables in $\bar{y}$ are different.*

Proof. Please see Section 5.1. □

4.2. Loop-separable program

Now we present our main result.

Definition 5 *(Loop-separable programs).* A program Π is *loop-separable* if for every loop pattern of form (15), one of the following cases holds:

Case 1: there is a pair (α_i, ρ_i), $1 \leqslant i < n$, such that $Body(\rho_i)$ can be separated into two parts B_h and B_b and
- $B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(\rho_i)$,
- $\alpha_{i+1} \in B_b$,
- $Var(\{\alpha_i\} \cup B_h) \cap Var(B_b) = \emptyset$;

Case 2: for every pair (α_i, ρ_i), $1 \leqslant i < n$, $Body(\rho_i)$ can be separated into two parts B_h and B_b and
- $B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(\rho_i)$,
- $\alpha_{i+1} \in B_b$,
- $Var(\{\alpha_i\} \cup B_h) \cap Var(B_b) = \bigcap_{j=1}^{n} Var(\alpha_j)$.

Example 7. Consider loop pattern lp_1 of program Π_4 and the pair $(P(x_1), r_2')$, where r_2' is

$$r_2(x/x_1, y/x_2, u/u_1, v/v_1) : P(x_1) \leftarrow P(x_2), S(x_1, u_1), \text{not}\, T(x_2, v_1).$$

We can separate $Body(r')$ into two parts: $B_h = \{S(x_1, u_1)\}$ and $B_b = \{P(x_2), \text{not}\, T(x_2, v_1)\}$. So lp_1 satisfies the condition of case 1 in Definition 5, and program Π_4 is loop-separable.

Consider loop patterns lp_1 and lp_2 of program Π_5. We can see that both of them satisfy the condition of case 2 in Definition 5, so program Π_5 is also loop-separable.

Note that the two cases in Definition 5 are not exclusive. There are loop patterns that satisfy the conditions of both cases. For instance, a loop pattern of form (15) could be both of case 1 and case 2 if $\bigcap_{j=1}^{n} Var(\alpha_j) = \emptyset$. Furthermore, given a program, it is possible that some of its loop patterns are of case 1 while others case 2.

Example 8. Consider the program Π_6:

$$r_1: \quad P(x) \leftarrow R_1(x),$$
$$r_2: \quad Q(x) \leftarrow R_2(x),$$
$$r_3: \quad P(x) \leftarrow Q(x), R_3(y),$$

r_4: $Q(x) \leftarrow P(x), R_4(y),$

r_5: $P(x) \leftarrow P(y), R_5(y).$

There are four forms of loop patterns:

$$lp_1: \begin{array}{l} \big(P(x_1), r_5(x/x_1, y/x_2)\big), \\ \big(P(x_2), r_5(x/x_2, y/x_3)\big), \\ \big(P(x_3), r_5(x/x_3, y/x_4)\big), \end{array} \tag{19}$$

$$lp_2: \begin{array}{l} \big(P(x_1), r_3(x/x_1, y/y_1)\big), \\ \big(Q(x_1), r_4(x/x_1, y/y_1)\big), \\ \big(P(x_1), r_3(x/x_1, y/y_1)\big), \end{array} \tag{20}$$

$$lp_3: \begin{array}{l} \big(Q(x_1), r_4(x/x_1, y/y_1)\big), \\ \big(P(x_1), r_3(x/x_1, y/y_1)\big), \\ \big(Q(x_1), r_4(x/x_1, y/y_1)\big), \end{array} \tag{21}$$

$$lp_4: \begin{array}{l} \big(Q(x_1), r_4(x/x_1, y/y_1)\big), \\ \big(P(x_1), r_3(x/x_1, y/x_2)\big), \\ \big(P(x_2), r_3(x/x_2, y/y_2)\big), \\ \big(Q(x_2), r_4(x/x_2, y/y_2)\big). \end{array} \tag{22}$$

As we can see that lp_4 is a loop pattern of case 1, lp_2 and lp_3 are loop patterns of case 2, and lp_1 is a loop pattern of both case 1 and case 2.

Theorem 2. *If a program is loop-separable, then it is first-order definable.*

According to Theorem 2, programs Π_4, Π_5 and Π_6 are first-order definable.

Furthermore, the problem of checking whether a given program is loop-separable is decidable, as indicated by the following theorem.

Theorem 3. *It is decidable to check whether a program is loop-separable.*

5. Proofs of the main theorems

The proofs of the main theorems (i.e. Theorems 2 and 3) are rather technical and tedious. However, the underlying ideas are simple. For Theorem 2, we first prove that it holds with the restriction of the UNA, then extend this result to the general case. The first step is of the most technically challenging. For this purpose, we need to relate the answer set semantics to so-called expansion tree, extended from the same technique in Datalog. Then, we show that for any loop-separable program, we can always pick up a finite set of expansion trees to capture all its answer sets. Based on these finite number of expansion trees, we can explicitly define a first-order sentence that exactly captures the original program.

Theorem 3 is a direct consequence of Proposition 7, which can be proven by showing that, under the operator $\sim$, there is a bound for the length of loop patterns of a given program.

5.1. Proofs of Proposition 7 and Theorem 3

We first prove Proposition 7 and Theorem 3, as the proof techniques are needed in proving Theorem 2. We need the following lemma about the length of a derivation path.

Lemma 1. *Let Π be a program. There exists an natural number N such that for any derivation path of form* (15)*, if $n > N$ then there exists i, j $(1 \leqslant i < j \leqslant n)$ such that $(\alpha_i, \rho_i) \sim (\alpha_j, \rho_j)$.*

Proof. Note that Π has only finite many rules, so the original statement follows immediately from the following one:

Let k be a natural number. There exists a natural number N_k such that for any set $\{\bar{t}_1, \ldots, \bar{t}_n\}$ of tuples of length k, if $n > N_k$ then there exists i, j $(1 \leqslant i < j \leqslant n)$ such that $\bar{t}_i \sim \bar{t}_j$.

In other words, there exists a bound for a set of term tuples of fixed length if there are no two similar term tuples in it.

We first ignore all constants. Now we divide the set of term tuples into categories such that each category is a partition of the k terms. That is, for any two term tuples $\bar{t}$ and $\bar{t}'$ in the same category, for every pair i, j $(1 \leqslant i \neq j \leqslant k)$, $t_i = t_j$ if and only if $t'_i = t'_j$. The number of categories is finite since k is a fixed number. More specifically, the number of all categories is exactly B_k, the k-th Bell number, which can be understood as the number of equivalence relations on a set with k members. Clearly, two term tuples in different categories are not similar. We now prove that there exists a bound for any of the categories if there are no two similar term tuples in it. Without loss of generality, we only consider the case that all terms in the term tuple are distinct. The other cases can be obtained in a similar way.

In this category, for any term tuple $\bar{t}$, there does not exist t_i and t_j such that $t_i = t_j$. Let T_k be the maximal number of term tuples of length k in this category such that there are no two similar term tuples in it. Clearly, $T_1 = 1$. Now consider to calculate T_k for $k > 1$. Suppose there are no two term tuples that are similar in this category. Let $\bar{t} = (x_1, \ldots, x_k)$ be a term tuple and $x_i \neq x_j$ $(1 \leqslant i \neq j \leqslant k)$. If another tuple $\bar{t}'$ is not similar to $\bar{t}$, then there must exist i $(1 \leqslant i \leqslant k)$ such that $x_i \in \bar{t}'$ and x_i is not in the i-th position in $\bar{t}'$. Without loss of generality, assume that x_1 is in the k-th position of $\bar{t}'$. Then, consider all the term tuples in this category such that x_1 is in the k-th position. The number of such term tuples is less or equal than T_{k-1}. Otherwise, there exist two term tuples that are similar. Thus, we have

$$T_1 = 1,$$
$$T_k \leqslant k(k-1)T_{k-1} + 1, \quad k > 1.$$

Here, k in $k(k-1)T_{k-1}$ means that there are k terms, $k-1$ means that these variables must be in a different position, and T_{k-1} means that, as discussed above, there are at most T_{k-1} term tuples by fixing a term in a particular position. Hence, T_k is bounded. In fact,

$$T_k \leqslant k!(k-1)! \times \sum_{1 \leqslant i \leqslant k} \frac{1}{i!(i-1)!}.$$

The above proof shows that there exists a bound when ignoring all constants. When considering constants, this statement still holds since the set of constants is finite. In fact, we can divide the set of term tuples into m groups that contain m different constants, $0 \leqslant m \leqslant C$, where C is the number of all constants. Then, each group has a bound if there does not exist two substations "similar". The above case is for $m = 0$. The proof can be easily extended to an arbitrary m. □

Proof of Proposition 7. By Lemma 1, given a program Π, there are natural numbers N_1 and N_2, such that for every loop pattern S of Π, the length of S is less than N_1, and thus, the number of variables in S is less than N_2. Let $v_1, \ldots, v_{N_2}$ be N_2 variables not in Π, and $\mathcal{S}_l$ be all the possible loop patterns of Π using these variables.

We can see that $\mathcal{S}_l$ is finite, and for every loop pattern S of Π, there is a loop pattern $S' \in \mathcal{S}$ and $S = S'\theta$, where $\theta = (\bar{x}/\bar{y})$ is a substitution, and all variables in $\bar{y}$ are different. □

Proof of Theorem 3. The set of loop patterns in $\mathcal{S}_l$ as defined in Proposition 7 is finite, and it is sufficient to check if the condition in Definition 5 holds for each loop pattern in $\mathcal{S}_l$. □

5.2. Correspondence between answer set and expansion tree

Now we extend the notion of expansion tree, introduced in Datalog [3], to first-order answer set program with extensional database, and show how it is related to the answer set semantics. More precisely, we show that a structure $\mathcal{A}$ is an answer set of a given program Π if and only if $\mathcal{A}$ is supported by a set of expansion trees of Π. In this subsection and the next, unless stated otherwise, we assume that all the structures considered are models of Σ_{una}.

Definition 6. An *expansion tree* T of a program Π is a (finite) tree such that

- the nodes of T are pairs of the form (α, ρ), where α is an atom and $\rho = r\theta$ such that $r \in \Pi$ is a rule, θ a substitution, and $\alpha = Head(\rho)$;
- for any node (α, ρ), let $\beta_1, \ldots, \beta_i$ be all the intensional atoms in $Pos(\rho)$, then (α, ρ) has i children labeled with the atom $\beta_1, \ldots, \beta_i$.

In particular, a node (α, ρ) of an expansion tree T is a leaf of T if and only if all the atoms in $Pos(\rho)$ are either equality or extensional atoms.

Let T be an expansion tree of a program Π and w a node of T. We use α_w and ρ_w to denote the atom and the rule of w, and T_w the subtree of T whose root is w. Let θ be a substitution. We use $w\theta$ to denote the pair $(\alpha_w\theta, \rho_w\theta)$, and $T\theta$ the expansion tree obtained from T by replace every node w in T by $w\theta$. We also use $dep(T)$ to denote the depth of an expansion tree T, and $tree(\Pi)$ the (infinite) set of all expansion trees of Π.

Given an expansion tree T of a program Π, without loss of generality, we assume that variables in T and variables in Π are disjoint, and that for any node w, the variables in the body of ρ_w either occur in the head of ρ_w or they do not occur in any nodes of T except T_w. Note that every path in T is a derivation path of Π.

Given an expansion tree T, we use r_T to denote the rule whose head is the atom of the root of T, and the positive body and negative body are defined by:

$$Pos(r_T) = \bigcup \{ Pos(\rho_w) \mid w \text{ is a leaf of } T \},$$
$$Neg(r_T) = \bigcup \{ Neg(\rho_w) \mid w \text{ is a node of } T \}.$$

Given two expansion trees T_1 and T_2, we say that T_1 *is subsumed by* T_2 (or T_2 *subsumes* T_1) if the roots of T_1 and T_2 are labeled by the same atom and $Body(r_{T_2}) \subset Body(r_{T_1})$.

Definition 7. Let Π be a program, $\mathcal{A}$ a structure of $\tau(\Pi)$ and $\mathcal{T}$ a (possibly infinite) set of expansion trees. We say that $\mathcal{A}$ is *supported* by $\mathcal{T}$, if for every intensional predicate P and $\bar{a} \in P^{\mathcal{A}}$, there is an expansion tree $T \in \mathcal{T}$ and an assignment σ on $Var(T)$ over $\mathcal{A}$ such that $\sigma(\bar{t}) = \bar{a}$ and $(\mathcal{A}, \sigma) \models \widehat{Body_{r_T}}$, where $P(\bar{t})$ is the atom of the root of T.

The following two lemmas show the relationship between the answer sets and expansion trees.

Lemma 2. *Let Π be a program and $\mathcal{A}$ a structure of $\tau(\Pi)$ which is a model of $\widehat{\Pi}$. If $\mathcal{A}$ is an answer set of Π, then $\mathcal{A}$ is supported by $tree(\Pi)$.*

Proof. Assume that $\mathcal{A}$ is an answer set of Π. Let $\mathcal{A}_e$ be the structure of $\tau_{ext}(\Pi)$ such that $Dom(\mathcal{A}_e) = Dom(\mathcal{A})$ and $P^{\mathcal{A}_e} = P^{\mathcal{A}}$ for all extensional predicate $P \in \mathcal{P}_{ext}(\Pi)$. Also, let $M = \{P(\bar{a}) \mid P \in \mathcal{P}_{int}(\Pi) \text{ and } \bar{a} \in P^{\mathcal{A}}\}$. By Proposition 1, M is an answer set of the propositional program $\Pi_{\mathcal{A}_e}$, where $\Pi_{\mathcal{A}_e}$ is the instantiation of Π on $\mathcal{A}_e$.

By Definition 7, it is sufficient to show that for every $P(\bar{a}) \in M$, there is an expansion tree $T \in tree(\Pi)$ and an assignment σ on $Var(T)$ over $\mathcal{A}$ such that $P(\bar{a}) = \alpha[\sigma]$ and $(\mathcal{A}, \sigma) \models \widehat{Body_{r_T}}$, where α is the atom of the root of T.

From the definition of the answer set for propositional program, M is the minimal set which satisfies every rules in $GL_M(\Pi_{\mathcal{A}_e})$. We define $M_0 = \emptyset$ and $M_i = TP(M_{i-1})$ for $i > 0$, where TP is a map from a subset of $\mathcal{L}_{\mathcal{A}}$ to a subset of $\mathcal{L}_{\mathcal{A}}$ defined as following:

$$TP(S) = \{ pa \mid \text{there is a rule } pa \leftarrow pb_1, \ldots, pb_k \text{ in } GL_M(\Pi_{\mathcal{A}_e}) \text{ such that } \{pb_1, \ldots, pb_k\} \subseteq S \}.$$

$GL_M(\Pi_{\mathcal{A}_e})$ is positive, so there exists n such that $M_n = M_{n+1} = M$ [30]. We will show by induction that if $P(\bar{a}) \in M_i$, $0 \leqslant i \leqslant n$, then there is an expansion tree $T \in tree(\Pi)$ and an assignment σ on $Var(T)$ over $\mathcal{A}$ such that $P(\bar{a}) = \alpha[\sigma]$ and $(\mathcal{A}, \sigma) \models \widehat{Body_{r_T}}$, where α is the atom of the root of T.

For $i = 0$, the statement holds trivially. We assume that the statement holds for all $i < j$, and we will show next that it also holds for j.

Let $P(\bar{a}) \in M_j$. By the definition of $TP(M_j)$, there is a rule

$$P(\bar{a}) \leftarrow pb_1, \ldots, pb_k$$

in $GL_M(\Pi_{\mathcal{A}_e})$ such that $\{pb_1, \ldots, pb_k\} \subseteq M_{j-1}$. By the definitions of $\Pi_{\mathcal{A}_e}$ and $GL_M(\Pi_{\mathcal{A}_e})$, there is a rule

$$P(\bar{a}) \leftarrow pb_1, \ldots, pb_k, \text{not } pc_1, \ldots, \text{not } pc_l$$

in $\Pi_{\mathcal{A}_e}$, and there is a rule r:

$$P(\bar{t}) \leftarrow b_1, \ldots, b_k, b_{k+1}, \ldots, b_{k'}, \text{not } c_1, \ldots, \text{not } c_l, \text{not } c_{l+1}, \ldots, \text{not } c_{l'}$$

in Π and an assignment σ_r on $Var(r)$ over $\mathcal{A}$ such that:

- $P(\bar{t})[\sigma_r] = P(\bar{a})$;
- $b_1, \ldots, b_k$ are intensional atoms, and $b_m[\sigma_r] = pb_m$, $1 \leqslant m \leqslant k$;
- $b_{k+1}, \ldots, b_{k'}$ are either extensional atoms or equality atoms, and $(\mathcal{A}, \sigma_r) \models b_m$, $k < m \leqslant k'$;
- $c_1, \ldots, c_l$ are intensional atoms, and $c_m[\sigma_r] = pc_m$, $1 \leqslant m \leqslant l$;
- $c_{l+1}, \ldots, c_{l'}$ are either extensional atoms or equality atoms;
- $(\mathcal{A}, \sigma_r) \not\models c_m$, $1 < m \leqslant l'$.

By assumption, for every $1 \leqslant m \leqslant k$, there is an expansion tree $T_m \in tree(\Pi)$ and an assignment σ_m on $Var(T_m)$ over $\mathcal{A}$ such that $pb_m = \alpha_m[\sigma_m]$ and $(\mathcal{A}, \sigma_m) \models \widehat{Body_{r_{T_m}}}$, where α_m is the atom of the root of T_m. Without loss of generality, we assume $Var(T_{m_1}) \cap Var(T_{m_2}) = \emptyset$, $1 \leqslant m_1 \neq m_2 \leqslant k$.

We introduce a new variable v_c for each domain element $c \in Dom(\mathcal{A})$, and define a substitution θ_m for each T_m, $1 \leqslant m \leqslant k$, as following:

$$\theta_m = \{x/v_c \mid x \in Var(T_m) \text{ and } x \text{ is assigned to } c \text{ in } \sigma_m\}.$$

We can see that $pb_m = \alpha_m[\sigma_m] = \alpha_m\theta_m[\sigma'_m]$, and $(\mathcal{A}, \sigma_m) \models \widehat{Body_{r_{T_m}}}$ if and only if $(\mathcal{A}, \sigma'_m) \models \widehat{Body_{r_{T_m\theta_m}}}$, where α_m is the root of T_m and σ'_m is the assignment on $Var(T_m\theta_m)$ over $\mathcal{A}$ which assign every variable of the form v_c to the domain element c.

Let θ be the substitution

$$\theta = \{x/v_c \mid x \in Var(r) \text{ and } x \text{ is assigned to } c \text{ in } \sigma_r\}.$$

We then define an expansion tree T as following:

- the root of T is $(P(\bar{t})\theta, r\theta)$,
- $(P(\bar{t})\theta, r\theta)$ have k subtree: $T_1\theta_1, \ldots, T_k\theta_k$.

We can see that $b_1\theta, \ldots, b_k\theta$ are all the intensional atoms in $Pos(r\theta)$. For $1 \leqslant m \leqslant k$, the atom of the root of $T_m\theta_m$ is $\alpha_m\theta_m$, and $\alpha_m\theta_m = b_m\theta$ by noticing that $\alpha_m\theta_m[\sigma'_m] = pb_m$ and $pb_m = b_m[\sigma_r] = b_m\theta[\sigma'_m]$. So T is well defined.

All variables in T are of the form v_c where c is a domain element of $\mathcal{A}$. Let σ^* be the assignment on $Var(T)$ that assign every variable of the form v_c in $Var(T)$ to the domain element c. We can see that:

- $T\theta$ is an expansion tree by noticing that $T_1\theta_1, \ldots, T_k\theta_k$ are expansion trees and that $(P(\bar{t})\theta, r\theta)$ has k children which are labeled with the atoms $b_1\theta, \ldots, b_k\theta$;
- the root of $T\theta$ is labeled by atom $P(\bar{t})\theta$ and $P(\bar{a}) = P(\bar{t})\theta[\sigma^*]$ by noticing $P(\bar{a}) = P(\bar{t})[\sigma_r]$;
- $(\mathcal{A}, \sigma^*) \models \widehat{Body_{r_{T\theta}}}$ by noticing that $(\mathcal{A}, \sigma_m) \models \widehat{Body_{r_{T_m\theta_m}}}$ $(1 \leqslant m \leqslant k)$, $(\mathcal{A}, \sigma^*) \models b_m$ $(k < m \leqslant k')$, and $(\mathcal{A}, \sigma^*) \not\models c_m$ $(1 \leqslant m \leqslant l')$.

This completes the proof. □

Lemma 3. *Let Π be a program and $\mathcal{A}$ be a structure of $\tau(\Pi)$ which is a model of $\widehat{\Pi}$. If $\mathcal{A}$ is supported by a set of expansion trees $\mathcal{T}$, then $\mathcal{A}$ is an answer set of Π.*

Proof. Assume that $\mathcal{A}$ is a model of $\widehat{\Pi}$, and is supported by a set of expansion trees $\mathcal{T}$. By Theorem 1, it is sufficient to show that $\mathcal{A}$ is a model of $LF(\Pi)$.

Let L be a loop of Π The loop formula of L is the universal closure of

$$\bigvee_{\alpha \in L} \alpha \supset \bigvee_{\alpha \in L} ES(\alpha, L, \Pi).$$

Let σ be an arbitrary assignment on $Var(L)$ over $\mathcal{A}$. If $(\mathcal{A}, \sigma) \not\models \bigvee_{\alpha \in L} \alpha$, then $(\mathcal{A}, \sigma) \models \bigvee_{\alpha \in L} \alpha \supset \bigvee_{\alpha \in L} ES(\alpha, L, \Pi)$. Otherwise, there exists $\bar{a} \in P^{\mathcal{A}}$ such that $P(\bar{t}) \in L$ and $\bar{a} = \sigma(\bar{t})$. We will show that $(\mathcal{A}, \sigma) \models \bigvee_{\alpha \in L} ES(\alpha, L, \Pi)$ also holds in this case.

By assumption, $\mathcal{A}$ is supported by $\mathcal{T}$. So there exists an expansion tree $T \in \mathcal{T}$ and an assignment δ on $Var(T)$ over $\mathcal{A}$ such that $(\mathcal{A}, \delta) \models \widehat{Body_{r_T}}$ and $\delta(\bar{t}^*) = \bar{a}$, where $P(\bar{t}^*)$ is the atom of the root of T. We will show by induction on the structure of the expansion tree T that for every node $w = (P'(\bar{t}'), \rho)$ of T,

- $(\mathcal{A}, \delta) \models P'(\bar{t}')$ and
- for every loop L', if there is an atom $P'(\bar{t}'') \in L'$ and an assignment σ' on $Var(L')$ over $\mathcal{A}$ such that $\sigma'(\bar{t}'') = \delta(\bar{t}')$, then $(\mathcal{A}, \sigma') \models \bigvee_{\alpha \in L'} ES(\alpha, L', \Pi)$. Without loss of generality, we assume $Var(L') \cap Var(T) = \emptyset$.

(1) If $w = (P'(\bar{t}'), \rho)$ is a leaf, then $Body(\rho) \subseteq Body(r_T)$. We have $(\mathcal{A}, \delta) \models \widehat{Body_\rho}$ and $(\mathcal{A}, \delta) \models P'(\bar{t}')$ by noticing that $\mathcal{A}$ is a model of $\widehat{\Pi}$.

Let L' be a loop and $\bar{x}$ the tuple of variables in L'. Let σ' be an assignment on $\bar{x}$ over $\mathcal{A}$ such that there is an atom $P'(\bar{t}'') \in L'$ and $\sigma'(\bar{t}'') = \delta(\bar{t}')$. Also, let $\bar{y}$ be the tuple of the local variables of ρ, and σ'' be the assignment on $\bar{x} \cup \bar{y}$ over $\mathcal{A}$ such that $\sigma''(\bar{x}) = \sigma'(\bar{x})$ and $\sigma''(\bar{y}) = \delta(\bar{y})$.

We have

$$(\mathcal{A}, \sigma'') \models \widehat{Body_\rho} \wedge \bigwedge_{Q(\bar{s}) \in L',\ Q(\bar{s}') \in Pos(\rho)} \bar{s} \neq \bar{s}'$$

by noticing that there are only intensional atoms in L', and that the positive part of ρ are all extensional atoms. So, we have

$$(\mathcal{A}, \sigma') \models \exists \bar{y}\left(\widehat{Body_\rho} \wedge \bigwedge_{Q(\bar{s})\in L',\ Q(\bar{s'})\in Pos(\rho)} \bar{s} \neq \bar{s'}\right)$$

and

$$(\mathcal{A}, \sigma') \models \bigvee_{\alpha\in L'} ES(\alpha, L', \Pi).$$

(2) If $w = (P'(\bar{t}'), \rho)$ is not a leaf, then let $w_1, \ldots, w_n$ be the children of w. Assume that for every child $w_i = (Q_i(\bar{t}_i), \rho_i)$ of w, $1 \leqslant i \leqslant n$,

- $(\mathcal{A}, \delta) \models Q_i(\bar{t}_i)$ and
- for every loop L'', if there is an atom $Q_i(\bar{t}'_i) \in L''$ and an assignment σ'' such that $\sigma''(\bar{t}'_i) = \delta(\bar{t}_i)$, then $(\mathcal{A}, \sigma'') \models \bigvee_{\alpha\in L''} ES(\alpha, L'', \Pi)$.

By the definition of r_T, we have $Neg(\rho) \subseteq Neg(r_T)$. We also have $Pos(\rho) = \{Q_1(\bar{t}_1), \ldots, Q_n(\bar{t}_n)\}$, and $(\mathcal{A}, \delta) \models Q_i(\bar{t}_i)$, $1 \leqslant i \leqslant n$. So, we have $(\mathcal{A}, \delta) \models \widehat{Body_\rho}$ and $(\mathcal{A}, \delta) \models P'(\bar{t}')$. We still need to show that for every loop L', if there is an atom $P'(\bar{t}'') \in L'$ and an assignment σ' on $Var(L')$ such that $\sigma'(\bar{t}'') = \delta(\bar{t}')$, then $(\mathcal{A}, \sigma') \models \bigvee_{\alpha\in L'} ES(\alpha, L', \Pi)$.

If there is a child $w_i = (Q_i(\bar{t}_i), \rho_i)$ of w, and there is an atom $Q_i(\bar{t}'_i) \in L'$ such that $\sigma'(\bar{t}'_i) = \delta(\bar{t}_i)$, then $(\mathcal{A}, \sigma') \models \bigvee_{\alpha\in L'} ES(\alpha, L', \Pi)$ by induction hypotheses. Otherwise, for any two atoms $Q(\bar{s}) \in L'$ and $Q(\bar{s'}) \in Pos(\rho)$, we have $\sigma'(\bar{s}) \neq \delta(\bar{s'})$. Similarly to (1), let $\bar{y}$ be the tuple of the local variables of ρ, and σ'' be the assignment on $\bar{x} \cup \bar{y}$ over $\mathcal{A}$ such that $\sigma''(\bar{x}) = \sigma'(\bar{x})$ and $\sigma''(\bar{y}) = \delta(\bar{y})$. We have

$$(\mathcal{A}, \sigma'') \models \widehat{Body_\rho} \wedge \bigwedge_{Q(\bar{s})\in L',\ Q(\bar{s'})\in Pos(\rho)} \bar{s} \neq \bar{s'}.$$

So,

$$(\mathcal{A}, \sigma') \models \exists \bar{y}\left(\widehat{Body_\rho} \wedge \bigwedge_{Q(\bar{s})\in L',\ Q(\bar{s'})\in Pos(\rho)} \bar{s} \neq \bar{s'}\right)$$

and

$$(\mathcal{A}, \sigma') \models \bigvee_{\alpha\in L'} ES(\alpha, L', \Pi). \qquad \square$$

5.3. Finite set of expansion trees for loop-separable program

From Lemmas 2 and 3, a program Π can be defined by $\widehat{\Pi}$ and all its expansion trees. However, there might exist infinite number of expansion trees in general by even considering the equivalence under substitution. Fortunately, we can show that for loop-separable programs, we can always find a finite set of expansion trees which is equivalent to $tree(\Pi)$.

The key idea is that for loop-separable programs, the depth of their expansion trees can be bounded to some extent. That is, for any loop-separable program, there exists a natural number k such that for any expansion tree of the program whose depth is greater than k, we can always construct another expansion tree, which subsumes the original one and whose depth is less than k. Roughly speaking, any large expansion tree of a loop-separable program can be unfolded into a smaller one.

For this purpose, we need to show some propositions of derivation paths and loop patterns. A derivation path of form (15) is called a *base pattern* if atoms in $Pos(\rho_n)$ are either equality or extensional atoms, and $\rho_i \nsim \rho_j$ $(1 \leqslant i < j \leqslant n)$. In the following, we will define how a derivation path is *extended* by a loop pattern, and show the relationship between the loop patterns and the derivation paths in expansion trees.

Definition 8. Let $S = (w_1, \ldots, w_n)$ be a derivation path, $S' = (w'_1, \ldots, w'_m)$ a loop pattern, and $Var(S) \cap Var(S') = \emptyset$. We say that S can be *extended* by S' if there is a node w_i $(1 \leqslant i \leqslant n)$ such that $w_i \sim w'_1$. An *extension* of S by S' is a sequence of pairs:

$$w_1, \ldots, w_{i-1}, w'_1\theta_1, \ldots, w'_m\theta_1, w_{i+1}\theta_2, \ldots, w_n\theta_2 \tag{23}$$

where

- $(\bar{x}/\bar{x'})$ is the witness of $w_i \sim w'_1$;
- $(\bar{y_1}/\bar{y_m})$ is the witness of $w'_1 \sim w'_m$;

- $\theta_1 = \bar{x}'/\bar{x} \cup \bar{z}'/\bar{z}$, where $\bar{z}'$ is the tuple of variables $Var(S') \setminus \bar{x}'$, $\bar{z}$ is a tuple of new variables not in $Var(S) \cup Var(S')$, and all variables in $\bar{z}$ are different;
- $\theta_2 = \{u/v \mid$ there is a variable v' such that $u/v' \in (\bar{x}/\overline{x'})$, $v'/v'' \in (\overline{y_1}/\overline{y_m})$ and $v''/v \in \theta_1\}$.

Proposition 8. *Let $S = (w_1, \ldots, w_n)$ be a derivation path and $S' = (w'_1, \ldots, w'_m)$ a loop pattern such that $Var(S) \cap Var(S') = \emptyset$. If S^* be an extension of S by S', then S^* is also a derivation path, $Var(S^*) \cap Var(S') = \emptyset$ and the length of S^* is $n + m - 1$.*

Proof. Let S^* be an extension of S by S' of form (23). By Definition 8, we have $w_i = w'_1(\overline{x'}/\bar{x}) = w'_1\theta_1$ and $w'_m\theta_1 = w_i\theta_2$, so S^* is also a derivation path.

Consider the substitution θ_1. All variables in S' are substitute to distinct new variables, so we have $Var(S^*) \cap Var(S') = \emptyset$. It is obvious that the length of S^* is $n + m - 1$. □

Given a derivation path S and loop patterns $S_1, \ldots, S_n$. We also say a derivation path S' is an *extension* of S by $S_1, \ldots, S_n$ if there exist derivation paths $S'_0(= S), S'_1, \ldots, S'_n(= S')$ such that for $1 \leqslant i \leqslant n$, S'_i is an extension of S'_{i-1} by S_i.

Proposition 9. *Let Π be a program and T an expansion tree of Π. Also, let $\mathcal{S}_l$ be the set of loop patterns as mentioned in Proposition 7 and assume that $Var(T) \cap Var(\mathcal{S}_l) = \emptyset$. If $S = (w_1, \ldots, w_n)$ is a path in T such that w_1 is the root of T and w_n a leaf, then S is either a base pattern or an extension of S_0 by $S_1, \ldots, S_m$, where S_0 is a base pattern and $S_1, \ldots, S_m$ are loop patterns such that $S_i \in \mathcal{S}_l$, $1 \leqslant i \leqslant m$.*

Proof. If $w_i \nsim w_j$ $(1 \leqslant i < j \leqslant n)$, then S is a base pattern. Otherwise, there exists w_i, w_j $(1 \leqslant i < j \leqslant n)$ in S such that $w_i \sim w_j$.

By Proposition 7, if there exists a base pattern S_0 and loop patterns $S'_1, \ldots, S'_m$ such that S is an extension of S_0 by $S'_1, \ldots, S'_m$, then there always exist loop patterns $S_1, \ldots, S_m$, such that $S_i \in \mathcal{S}_l$ $(1 \leqslant i \leqslant m)$ and S is an extension of S_0 by $S_1, \ldots, S_m$. In the following, we will show that if S is not a base pattern, then there exists a derivation path S^*_1 and a loop pattern S^*_2 such that the length of S^*_1 is less than the length of S and S is an extension of S^*_1 by S^*_2.

Without loss of generality, we assume that $w_{i'} \nsim w_{j'}$ for every $i \leqslant i' < j' \leqslant j$ unless $i = i'$ and $j = j'$. Let $S^*_1 = (w_1, \ldots, w_i, w_{j+1}(\overline{y_1}/\overline{x_1}), \ldots, w_n(\overline{y_1}/\overline{x_1}))$, where $(\overline{x_1}/\overline{y_1})$ is the witness of $w_i \sim w_j$. Also, let $S^*_2 = (w_i(\overline{x_2}/\overline{y_2}), \ldots, w_j(\overline{x_2}/\overline{y_2}))$, where $\overline{x_2}$ is the tuple of all variables in $\{w_i, \ldots, w_j\}$, $\overline{y_2}$ is a tuple of new variables not in T and all variables in $\overline{y_2}$ are different.

S^*_1 is a derivation path by noticing that $w_i = w_j(\overline{y_1}/\overline{x_1})$ and all variables in $\overline{x_1}$ do not occur in $w_{j+1}, \ldots, w_n$. S^*_2 is a loop pattern and $Var(S^*_1) \cap Var(S^*_2) = \emptyset$ by noticing that $\overline{y_2}$ is a tuple of new variables. Furthermore, by Definition 8 and Proposition 8, S is an extension of S^*_1 by S^*_2, and the length of S^*_1 is less than the length of S. □

In the following, we will show that some properties of a derivation path still keep unchanged when it is extended. We introduce one more notion here. Given a derivation path S of form (15), we use $NS(S)$ to denote the number of pair (α_i, ρ_i), $1 \leqslant i < n$, such that $Body(\rho_i)$ can be separated into two parts B_h and B_b and

- $B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(\rho_i)$,
- $\alpha_{i+1} \in B_b$,
- $Var(\{\alpha_i\} \cup B_h) \cap Var(B_b) = \emptyset$.

Proposition 10. *Let $S = (w_1, \ldots, w_n)$ be a derivation path, $S' = (w'_1, \ldots, w'_m)$ a loop pattern and $Var(S) \cap Var(S') = \emptyset$. Let S^* be the extension of S by S' of form (23), then*

(1) *if $(w'_1, \ldots, w'_m)$ is a loop pattern of case 1 in Definition 5, and $w'_j = (\alpha'_j, \rho'_j)$ is the pair such that $Body(\rho'_j)$ can be separated into two parts B_h and B_b and*
   - *$B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(\rho'_j)$,*
   - *$\alpha'_{j+1} \in B_b$,*
   - *$Var(\{\alpha'_j\} \cup B_h) \cap Var(B_b) = \emptyset$,*

   then $(w'_1\theta_1, \ldots, w'_m\theta_1)$ is also a loop pattern of case 1, and $Body(w'_j)\theta_1$ can be separated into two parts $B_h\theta_1$ and $B_b\theta_1$ and
   - *$B_h\theta_1 \cap B_b\theta_1 = \emptyset$ and $B_h\theta_1 \cup B_b\theta_1 = Body(\rho_j)\theta_1$,*
   - *$\alpha'_{j+1}\theta_1 \in B_b\theta_1$,*
   - *$Var(\{\alpha'_j\theta_1\} \cup B_h\theta_1) \cap Var(B_b\theta_1) = \emptyset$;*

(2) *if $(w'_1, \ldots, w'_m)$ is a loop pattern of case 2 in Definition 5, then $(w'_1\theta_1, \ldots, w'_m\theta_1)$ is also a loop pattern of case 2 in Definition 5;*

(3) *$NS(S^*) = NS(S) + NS(S')$.*

Proof. Statements (1) and (2) hold by noticing that a variable is always replaced by a distinct new one in substitutions θ_1 and θ_2.

For statement (3), we have $w_i = w'_1(\overline{x'}/\bar{x}) = w'_1\theta_1$ and $w'_m\theta_1 = w_i\theta_2$. So,

$$\begin{aligned} NS(S) &= NS\big((w_1,\ldots,w_i)\big) + NS\big((w_i,\ldots,w_n)\big) \\ &= NS\big(\big(w_1,\ldots,w_{i-1},w'_1\theta_1\big)\big) + NS\big((w_i\theta_2,\ldots,w_n\theta_2)\big) \\ &= NS\big(\big(w_1,\ldots,w_{i-1},w'_1\theta_1\big)\big) + NS\big(\big(w'_m\theta_1,\ldots,w_n\theta_2\big)\big), \end{aligned}$$

and

$$\begin{aligned} NS\big(S^*\big) &= NS\big(\big(w_1,\ldots,w_{i-1},w'_1\theta_1\big)\big) + NS\big(S'\theta_1\big) + NS\big(\big(w'_m\theta_1,\ldots,w_n\theta_2\big)\big) \\ &= NS(S) + NS\big(S'\theta_1\big) \\ &= NS(S) + NS\big(S'\big). \qquad \square \end{aligned}$$

In the following two propositions, we will show that for some large expansion tree T, we can always find another smaller one T^* such that T is subsumed by T^* and the depth of T^* is less than T.

Proposition 11. *Given a program Π and an expansion tree T of Π, let $S = (w_1,\ldots,w_n)$ be a path in T such that w_1 is the root of T and w_n a leaf. If S is an extension of a base pattern S_0 by loop patterns $S_1,\ldots,S_m$ and there exists a loop pattern S^* of case 1 in Definition 5 such that S^* occurs in $S_1,\ldots,S_m$ more than once, then there is an expansion tree T^* such that T is subsumed by T^*.*

Proof. S^* is loop pattern of case 1 in Definition 5. Let $w = (\alpha_k, \rho_k)$ be the pair in S^* such that $Body(\rho_k)$ can be separated into two parts B_h and B_b and

- $B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(\rho_k)$,
- $\alpha_{k+1} \in B_b$,
- $Var(\{\alpha_k\} \cup B_h) \cap Var(B_b) = \emptyset$.

From (3) in Proposition 10, there exist two nodes w_i, w_j in S such that

- $w_i = (\alpha_i, \rho_k\theta_i)$ and $w_j = (\alpha_j, \rho_k\theta_j)$;
- for two different variables x_1, x_2, $x_1/y_1 \in \theta_i$ and $x_2/y_2 \in \theta_i$, then y_1, y_2 are two different variables;
- for two different variables x_1, x_2, $x_1/y_1 \in \theta_j$ and $x_2/y_2 \in \theta_j$, then y_1, y_2 are two different variables.

So, $Body(\rho_k)\theta_i$ can be also separated into two parts $B_h\theta_i$ and $B_b\theta_i$ and

- $B_h\theta_i \cap B_b\theta_i = \emptyset$ and $B_h\theta_i \cup B_b\theta_i = Body(\rho_k\theta_i)$,
- $\alpha_{i+1}\theta_i \in B_b\theta_i$,
- $Var(\{\alpha_i\theta_i\} \cup B_h\theta_i) \cap Var(B_b\theta_i) = \emptyset$.

Let θ^* be the substitution:

$$\theta^* = \big\{x/t \;\big|\; x \in V^*,\ x/t \in \theta_j\big\} \cup \big\{x/t \;\big|\; x \notin V^*,\ x/t \in \theta_i\big\},$$

where V^* is the set of variables in B_b. Consider the rule $\rho_k\theta^*$, we have

- $Head(\rho_k\theta^*) = \alpha_i$;
- for every literal $l \in Body(\rho_k)$, if $l\theta_i \in B_h\theta_i$, then $l\theta^* = l\theta_i \in Body(\rho_k\theta_i)$;
- for every literal $l \in Body(\rho_k)$, if $l\theta_i \in B_b\theta_i$, then $l\theta^* = l\theta_j \in Body(\rho_k\theta_j)$.

We can construct T^* as shown in Fig. 1. Let T^* be the expansion tree obtained from T by

- keep the nodes not in the subtree of w the same as T;
- replace the node w by $w^* = (\alpha_i, \rho_k\theta^*)$;
- for every intensional atom $\beta^* \in Body(\rho_k\theta^*)$, if $\beta \in Body(\rho_k)$ is the atom such that $\beta\theta_i = \beta^* \in B_h\theta_i$ then $(\alpha_w, \rho_k\theta^*)$ has a child labeled by atom β^*, and copy all the node in the subtree of $\beta\theta_i$ in T_{w_i} to T^*;
- for every intensional atom $\beta^* \in Body(r\theta^*)$, if $\beta \in Body(\rho_k)$ is the atom such that $\beta\theta_i \in B_b\theta_i$ and $\beta\theta_j = \beta^*$, then $(\alpha_w, \rho_k\theta^*)$ has a child labeled by atom β^*, and copy all the node in the subtree of $\beta\theta_j$ in T_{w_j} to T^*.

From the construction of T^*, we can see that the roots of T and T^* are labeled by the same atom, and $Body(r_{T^*}) \subset Body(r_T)$, so T is subsumed by T^*. $\square$

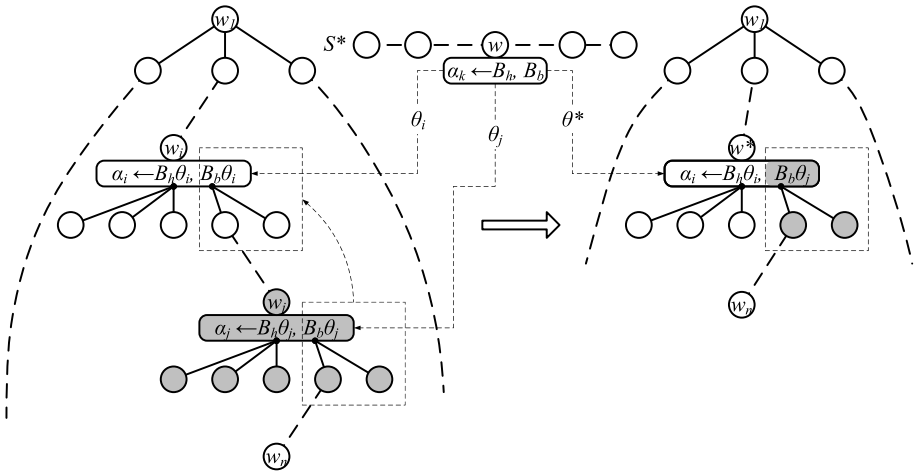

Fig. 1. The proof of Proposition 11.

Proposition 12. *Given a program Π and an expansion tree T of Π, let $S = (w_1, \ldots, w_n)$ be a path in T such that w_1 is the root of T and w_n a leaf. If there exist w_i and w_j in S such that $(w_i, \ldots, w_j)$ is a loop pattern of case 2 in Definition 5, then there is an expansion tree T^* such that T is subsumed by T^*.*

Proof. $(w_i, \ldots, w_j)$ is a loop pattern of case 2 in Definition 5. Let $w_i = (\alpha_i, r(\bar{x}/\bar{t}_i))$, $w_j = (\alpha_j, r(\bar{x}/\bar{t}_j))$. From case 2 of Definition 5, $r(\bar{x}/\bar{t}_i)$ can be separated into two parts B_h and B_b such that

- $B_h \cap B_b = \emptyset$ and $B_h \cup B_b = Body(r(\bar{x}/\bar{t}_i))$,
- $\alpha_{i+1} \in B_b$,
- $Var(\{\alpha_i\} \cup B_h) \cap Var(B_b) = \bigcap_{k=i}^{j} Var(\alpha_{w_k})$.

Let θ^* be the substitution:

$$\theta^* = \{x/t \mid x \in V^*, x/t \in \bar{x}/\bar{t}_j\} \cup \{x/t \mid x \notin V^*, x/t \in \bar{x}/\bar{t}_i\},$$

where V^* is the set of variables in B_b and not in $\bigcap_{k=i}^{j} Var(\alpha_{w_k})$.

Consider the rule $r\theta^*$, we have

- $Head(r\theta^*) = \alpha_{w_i}$;
- for every literal $l \in Body(r)$, if $l(\bar{x}/\bar{t}_i) \in B_h$, then $l\theta^* = l(\bar{x}/\bar{t}_i) \in Body(r(\bar{x}/\bar{t}_i))$;
- for every literal $l \in Body(r)$, if $l(\bar{x}/\bar{t}_i) \in B_b$, then $l\theta^* = l(\bar{x}/\bar{t}_j) \in Body(r(\bar{x}/\bar{t}_j))$.

We can construct T^* as shown in Fig. 2. Let T^* be the expansion tree obtained from T by

- keep the nodes not in the subtree of w_i the same as T;
- replace the node w_i by $(\alpha_{w_i}, r\theta^*)$;
- for every intensional atom $\beta^* \in Body(r\theta^*)$, if $\beta \in Body(r)$ is the atom such that $\beta(\bar{x}/\bar{t}_i) = \beta^* \in B_h$ then $(\alpha_{w_i}, r\theta^*)$ has a child labeled by atom β^*, and copy all the node in the subtree of $\beta(\bar{x}/\bar{t}_i)$ in T_{w_i} to T^*;
- for every intensional atom $\beta^* \in Body(r\theta^*)$, if $\beta \in Body(r)$ is the atom such that $\beta(\bar{x}/\overline{y_i}) \in B_b$ and $\beta(\bar{x}/\bar{t}_j) = \beta^*$, then $(\alpha_{w_i}, r\theta^*)$ has a child labeled by atom β^*, and copy all the node in the subtree of $\beta(\bar{x}/\bar{t}_j)$ in T_{w_j} to T^*.

From the construction of T^*, we can see that the roots of T and T^* are labeled by the same atom, and $Body(r_{T^*}) \subset Body(r_T)$, so T is subsumed by T^*. □

To end this section, we draw the conclusion that every loop-separable program can be captured by a finite set of its expansion trees.

Lemma 4. *Given a loop-separable program Π, there exists a finite set of expansion trees $\mathcal{T}$ such that for any structure $\mathcal{A}$ of $\tau(\Pi)$ which is a model of $\widehat{\Pi}$, if $\mathcal{A}$ is supported by tree(Π), then $\mathcal{A}$ is supported by $\mathcal{T}$.*

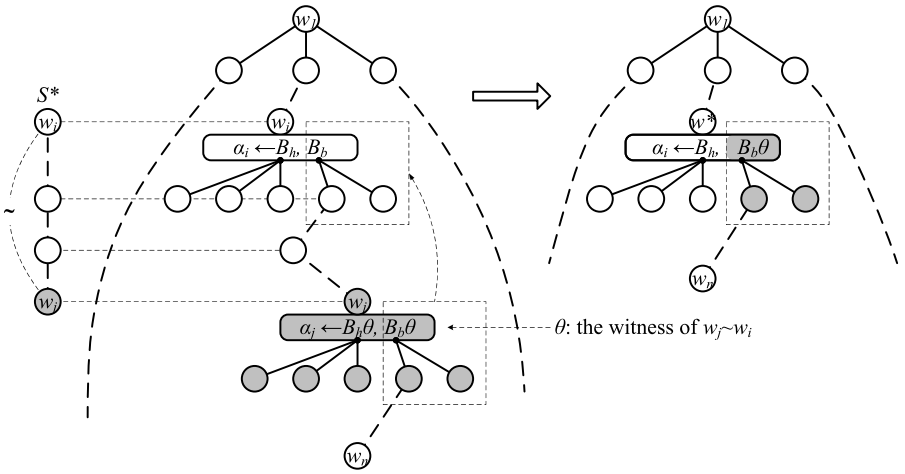

Fig. 2. The proof of Proposition 12.

Proof. Let $\mathcal{S}_l$ be the set of loop patterns as mentioned in Proposition 7. Let $\mathcal{S}_l = \mathcal{S}_{l1} \cup \mathcal{S}_{l2}$, where $\mathcal{S}_{l1}$ is the set of loop patterns of case 1 in Definition 5 and $\mathcal{S}_{l2}$ is the set of loop patterns of case 2 and not case 1 in Definition 5. Let

$$N_1 = (num_{ns} + max_{bn} + 1) \times (N + 1) + max_{bl},$$

where

- $num_{ns} = \sum_{S \in \mathcal{S}_{l1}} NS(S)$,
- N is the number mentioned in Lemma 1,
- max_{bl} is the maximum length of base patterns of Π,
- max_{bn} is the maximum $NS(S_b)$ when S_b is a base pattern.

By Lemma 1, max_{bl} and max_{bn} always exist. Let N_2 be the maximum number of variables in an expansion tree whose depth is less than or equal to N_1. Let $v_1, \ldots, v_{N_2}$ be N_2 variables not in Π, and $\mathcal{T}$ be the set of all the expansion trees using these variables whose depth is less than or equal to N_1.

It is obvious that $\mathcal{T}$ is finite, and for every expansion trees T such that T do not share variables with $\mathcal{T}$ and $dep(T) \leqslant N_1$, there is an expansion tree $T' \in \mathcal{T}$ and a substitution $\bar{x}/\bar{y}$ such that $T = T'(\bar{x}/\bar{y})$, where $\bar{x}$ is the tuple of all variables in T'.

Let $T \in tree(\Pi)$ be an expansion tree and $dep(T) > N_1$. It is sufficient to show that there exists an expansion tree T' such that T is subsumed by T' and $dep(T') \leqslant N_1$.

Let $S = (w_1, \ldots, w_n)$ be a path in T, where w_1 is the root of T, w_n is a leaf of T, and $n = dep(T)$. We can see that

(1) either $NS(S) > num_{ns} + max_{bn}$, or
(2) there are w_{k_1}, w_{k_2} in S such that $k_2 - k_1 > N$ and $NS(\{w_{k_1+1}, \ldots, w_{k_2}\}) = 0$.

We will show that in both cases, there exists an expansion tree T^* such that T is subsumed by T^*.

(1) By Proposition 9, assume that S is an extension of S_0 by $S_1, \ldots, S_m$, where S_0 is a base pattern and $S_1, \ldots, S_m$ are loop patterns such that $S_i \in \mathcal{S}_l$, $1 \leqslant i \leqslant m$. By Proposition 10, we have $NS(S) = \sum_{k=0}^{m} NS(S_k)$. We already have $NS(S) > num_{ns} + max_{bn}$. So there exists a loop pattern S^* such that S^* is a loop pattern of case 1 in Definition 5 and S^* occurs in $S_1, \ldots, S_m$ more than once. By Proposition 11, we can construct an expansion tree T^* such that T is subsumed by T^*.

(2) From Lemma 1, there are two nodes $w_i, w_j \in S$ $(k_1 < i < j \leqslant k_2)$ such that $w_i \sim w_j$ and $NS((w_i, \ldots, w_j)) = 0$. Without loss of generality, assume that $w_{i'} \nsim w_{j'}$ $(i \leqslant i' < j' \leqslant j)$ unless $i = i'$ and $j = j'$. We can see that $(w_i, \ldots, w_j)$ is a loop pattern of case 2 in Definition 5. By Proposition 12, we can construct an expansion tree T^* such that such that T is subsumed by T^*.

Repeat above construction. We can always find an expansion tree T' such that T is subsumed by T' and $dep(T') \leqslant N_1$. □

5.4. Proof of Theorem 2

Now, we face to the proof of our main theorem, namely Theorem 2. We first consider the case with the restriction of the UNA, and then release the condition to the general case.

Lemma 5. *If a program Π is loop-separable, then Π is first-order definable under UNA. That is, there exists a first-order sentence ψ on vocabulary $\tau(\Pi)$ such that for every structure $\mathcal{A}$ of $\tau(\Pi)$ which is a model of Σ_{una}, it is an answer set of Π if and only if $\mathcal{A}$ is a model of ψ.*

Proof. Let $\mathcal{T}$ be the set of expansion trees mentioned in Lemma 4. From Lemmas 2, 3 and 4, Π is defined by the conjunction of $\widehat{\Pi}$ and the following sentences:

$$\forall \bar{x}\left(P(\bar{x}) \supset \bigvee_{T\in\mathcal{T}^*} \widehat{Body_{r_T}} \wedge \bar{x} = \bar{t}\right)$$

for every predicate $P \in \mathcal{P}_{int}(\Pi)$, where $\mathcal{T}^*$ is the set of expansion trees in $\mathcal{T}$ whose root is labeled by the atom of the form $P(\bar{t})$. □

Clearly, if a program is first-order definable then it is first-order definable under UNA. However, the converse does not hold in general.

Example 9. We consider a program Π_7 consisting of the following rules:

$$T(x, y) \leftarrow E(x, y), \text{not } E(x, x), \text{not } E(y, y),$$

$$T(x, z) \leftarrow T(x, y), T(y, z),$$

$$P(a, b) \leftarrow a = b, \tag{24}$$

$$P(a, b) \leftarrow \text{not } P(a, b), \tag{25}$$

where $\mathcal{C}(\tau(\Pi_7)) = \{a, b\}$, $\mathcal{P}_{ext}(\Pi_7) = \{E\}$ and $\mathcal{P}_{int}(\Pi) = \{T, P\}$.

For any structure $\mathcal{A}$ which is a model of Σ_{una}, we can see that $\mathcal{A}$ is not an answer set of Π by noticing that rule (24) is never triggered and rule (25) is a contradiction. Π has no answer set which is a model of Σ_{una}, and is defined by $\bot$ under UNA.

If we also consider structures where a and b are mapped to the same domain element, we can see that Π is not first-order definable. Otherwise, the program Π_1 in Example 1 would be first-order definable.[3]

Fortunately, for loop-separable programs, they are always first-order definable even without the restriction of the UNA.

Given a vocabulary τ, let $\mathcal{F}_\tau$ be the set of all functions f, where

- the domain of f is $\mathcal{C}(\tau)$, and the range of f is a subset of $\mathcal{C}(\tau)$;
- if $c \in \mathcal{C}(\tau)$ is in the range of f, then $f(c) = c$.

Given a program Π and $f \in \mathcal{F}_{\tau(\Pi)}$ a function, let Π_f be the program obtained by replace each occurrence of c in Π by $f(c)$ for all $c \in \mathcal{C}(\tau(\Pi))$. Note that the only difference between vocabularies $\tau(\Pi)$ and $\tau(\Pi_f)$ is the set of constants. The set of constants of $\tau(\Pi_f)$ is a subset of that of $\tau(\Pi)$.

Proposition 13. *Let Π be a program, and $f \in \mathcal{F}_{\tau(\Pi)}$ a function. If Π is a loop-separable program, then Π_f is a loop-separable program.*

Proof. Directly from the definition of loop-separable program. □

The following proposition explicitly defines a sentence that captures a loop-separable program without the restriction of the UNA.

Proposition 14. *Let Π be a loop-separable program. Π is defined by*

$$\bigvee_{f\in\mathcal{F}_{\tau(\Pi)}} \left(\psi_{\Pi_f} \wedge \Sigma_{una}\big(\tau(\Pi_f)\big) \wedge \bigwedge_{c\in\mathcal{C}(\tau(\Pi))} \big(f(c) = c\big)\right), \tag{26}$$

where Π_f is defined by ψ_{Π_f} under UNA.

Proof. Let $\mathcal{A} = (A, c_1^{\mathcal{A}}, \ldots, c_m^{\mathcal{A}}, R_1^{\mathcal{A}}, \ldots, R_n^{\mathcal{A}})$ be a structure of $\tau(\Pi)$. It is sufficient to show that:

[3] For the proof of the indefinability of Π_1, please refer to Example 3 and Proposition 2 in [5]. The basic ideas of these two examples are quite similar.

(i) if $\mathcal{A}$ is an answer set of Π, then there is a function $f \in \mathcal{F}_{\tau(\Pi)}$ such that $\mathcal{A}$ is a model of $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f)) \wedge \bigwedge_{c\in\mathcal{C}(\tau(\Pi))}(f(c) = c)$;
(ii) if there is a function $f \in \mathcal{F}_{\tau(\Pi)}$ such that $\mathcal{A}$ is a model of $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f)) \wedge \bigwedge_{c\in\mathcal{C}(\tau(\Pi))}(f(c) = c)$, then $\mathcal{A}$ is an answer set of Π.

(i) Let $f \in \mathcal{F}_{\tau(\Pi)}$ be a function such that $f(c_1) = f(c_2)$ if and only if $c_1^{\mathcal{A}} = c_2^{\mathcal{A}}$, where c_1 and c_2 are constants in $\tau(\Pi)$. Let $\mathcal{A}' = (A, c_1^{\mathcal{A}'}, \ldots, c_{m'}^{\mathcal{A}'}, R_1^{\mathcal{A}}, \ldots, R_n^{\mathcal{A}})$ be the structure of $\tau(\Pi_f)$, where $c_i^{\mathcal{A}'} = f(c_i)^{\mathcal{A}}$ for $1 \leqslant i \leqslant m'$. From the definition of f and $\mathcal{A}'$, $\mathcal{A}'$ is a model of $\Sigma_{una}(\tau(\Pi_f))$.

If $\mathcal{A}$ is an answer set of Π, then $\mathcal{A}'$ is an answer set of Π_f. Π_f is also a loop-separable program, and is defined by ψ_f under UNA, so $\mathcal{A}'$ is a model of $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f))$ by noticing that $\mathcal{A}'$ is model of $\Sigma_{una}(\tau(\Pi_f))$. Notice that $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f)) \wedge \bigwedge_{c\in\tau(\Pi)}(f(c) = c)$ is also a formula of vocabulary of $\tau(\Pi)$, so $\mathcal{A}$ is a model of $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f)) \wedge \bigwedge_{c\in\tau(\Pi)}(f_{\mathcal{A}}(c) = c)$.

(ii) Let $f \in \mathcal{F}_{\tau(\Pi)}$ be the function such that $\mathcal{A}$ is a model of $\psi_{\Pi_f} \wedge \Sigma_{una}(\tau(\Pi_f)) \wedge \bigwedge_{c\in\tau(\Pi)}(f(c) = c)$. Let $\mathcal{A}' = (A, c_1^{\mathcal{A}'}, \ldots, c_{m'}^{\mathcal{A}'}, R_1^{\mathcal{A}}, \ldots, R_n^{\mathcal{A}})$ be the structure of $\tau(\Pi_f)$, where $c_i^{\mathcal{A}'} = f(c_i)^{\mathcal{A}'}$ for each constant c_i in vocabulary $\tau(\Pi_f)$. We can see that $\mathcal{A}'$ is a model of $\Sigma_{una}(\Pi_f)$ and ψ_{Π_f}, so it is an answer set of Π_f, and thus $\mathcal{A}$ is an answer set of Π. □

Finally, Theorem 2 follows from Lemma 5 and Proposition 14.

Proof of Theorem 2. Lemma 5 shows that a loop-separable program is first-order definable under UNA, and Proposition 14 shows that it is also first-order definable without the restriction of Σ_{una}. □

6. Subclasses and related results

We identify some interesting subclasses of loop-separable program, and show some related results.

6.1. Programs with a finite complete set of loops

We show here that the set of programs with a finite complete set of loops is a subset of loop-separable programs.

Proposition 15. *If program Π has a finite complete set of loops, then Π is a loop-separable program.*

Proof. By Theorem 2 in [4], a program Π has a finite complete set of loops, if and only if for every loop L and two atoms $\alpha, \beta \in L$, $Var(\alpha) = Var(\beta)$.

Let $(\alpha_1, \rho_1), \ldots, (\alpha_m, \rho_m)$ be a loop pattern. We show first that for every $1 \leqslant i < j \leqslant m$, $Var(\alpha_i) = Var(\alpha_j)$. Otherwise, assume that $Var(\alpha_i) \neq Var(\alpha_j)$ for some $1 \leqslant i < j \leqslant m$. Let $\bar{x}/\bar{y}$ be the witness of $(\alpha_1, \rho_1) \sim (\alpha_m, \rho_m)$, $\theta = \bar{x}/\bar{y} \cup \bar{y}/\bar{x}$, and $L = \{\alpha_1, \ldots, \alpha_m\}$ be a set of atoms. We can see that $L \cup L\theta$ is a loop of Π by noticing that $\alpha_m = \alpha_1\theta$ and $\alpha_1 = \alpha_m\theta$. We still have $\{\alpha_i, \alpha_j\} \subseteq L \cup L\theta$ and $Var(\alpha_i) \neq Var(\alpha_j)$, which contradicts to Theorem 2 in [4].

Thus, for every rule ρ_i $(1 \leqslant i \leqslant m)$, ρ_i can be separated into two parts: $B_h = \emptyset$ and $B_b = Body(\rho_i)$. □

Example 10. We continue with the program Π_2 in Example 2. Now we consider Π_2 as a loop-separable program. For intensional predicate P, we need to consider the following two expansion trees:

$$T_1: \quad \big(P(y), r_3(x/y)\big),$$
$$T_2: \quad \big(P(y), r_1(x/y)\big) - \big(Q(y), r_4(x/y)\big)$$

and for Q, the following two expansion trees:

$$T_3: \quad \big(Q(y), r_4(x/y)\big),$$
$$T_4: \quad \big(Q(y), r_2(x/y)\big) - \big(P(y), r_3(x/y)\big).$$

Thus, Π_2 is defined by the conjunction of $\widehat{\Pi_2}$ and

$$\forall y \quad \big(P(y) \supset \big(R_3(y) \vee \big(R_2(y) \wedge R_4(y)\big)\big)\big),$$
$$\forall y \quad \big(Q(y) \supset \big(R_4(y) \vee \big(R_1(y) \wedge R_3(y)\big)\big)\big).$$

As we can see, a program with a finite complete set of loops can be defined by using loop formulas or by using expansion trees, and the formulas obtained by these two ways are quite different.

6.2. Separable on rules

It is obvious that "loop-separable" is not a modular property. Program $\Pi \cup \Pi'$ is possibly not loop-separable program, when both programs Π and Π' are loop-separable programs. However, if we consider the "separable" property on each rule of a program, we can specify a subclass of loop-separable programs.

A program Π is *rule-separable* if for every rule $r \in \Pi$, $Body(r)$ can be separated into two parts B_h and B_b such that $Var(B_h) \subseteq Var(Head(r))$ and $Var(B_b) \cap Var(Head(r)) = \emptyset$.

Proposition 16. *A rule-separable program is loop-separable.*

Proof. Let Π be a rule-separable program, and $S = (\alpha_1, \rho_1), \ldots, (\alpha_n, \rho_n)$ be a loop pattern of Π, then for every ρ_i $(1 \leqslant i \leqslant n)$, $Body(\rho_i)$ can be separated into two parts B_{h_i} and B_{b_i} such that $Var(B_{h_i}) \subseteq Var(Head(\rho_i))$ and $Var(B_{b_i}) \cap Var(Head(\rho_i)) = \emptyset$.

If there exists $1 \leqslant i < n$ such that $\alpha_{i+1} \in B_{b_i}$, then S is a loop pattern of case 1 in Definition 5.

Otherwise, for $1 \leqslant i < n$, we have $\alpha_{i+1} \in B_{h_i}$. By the definition of rule-separable program, we have $Var(\alpha_{i+1}) \subset Var(\alpha_i)$ for $1 \leqslant i < n$. By the definition of loop pattern, we have $\alpha_1 \sim \alpha_n$, and $Var(\alpha_1) = \cdots = Var(\alpha_n)$. So, S is a loop pattern of case 2 in Definition 5. □

We can see that if Π and Π' are two rule-separable programs, then $\Pi \cup \Pi'$ is also a rule-separable program.

A program is called a *unary program* if it has only unary predicates and no equality $(=)$. The program Π_6 in Example 8 is a unary program.

Proposition 17. *If Π is a unary program, then Π is rule-separable.*

Proof. For every rule $r \in \Pi$, the body of r can be separated into two parts B_h and B_b, where

$$B_h = \big\{\alpha \;\big|\; \alpha \in Body(r) \text{ and } Var(\alpha) = Var\big(Head(r)\big)\big\},$$

and

$$B_b = \big\{\alpha \;\big|\; \alpha \in Body(r) \text{ and } Var(\alpha) \neq Var\big(Head(r)\big)\big\}. \qquad \square$$

By Proposition 17, a unary program is first-order definable. If we consider our definition of answer set as a second-order formula, then this result is a special case of Theorem 8 in [7] (also see [24]), which shows that any second-order sentence that only contains unary predicates is always first-order definable.

6.3. Separable on loops

If we replace "loop patterns" by "loops" in Definition 5, we get another subclass of loop-separable program, as shown in the following proposition.

Proposition 18. *Let Π be a program. If for each loop L of Π, one of the following holds*:

(a) *for every cycle $\alpha_1, \ldots, \alpha_n (= \alpha_1)$ such that $\{\alpha_1, \ldots, \alpha_n\} = L$, there is α_i $(1 \leqslant i < n)$ such that for any rule $r \in \Pi$ and substitution θ, if $\alpha_i = Head(r\theta)$ and $\alpha_{i+1} \in Body(r\theta)$, then $Body(r\theta)$ can be separated into two parts B_h and B_b, such that*:
   - $B_h \cap B_b = \emptyset$ *and* $B_h \cup B_b = Body(r\theta)$;
   - $\alpha_{i+1} \in B_b$;
   - $Var(\{\alpha_i\} \cup B_h) \cap Var(B_b) = \emptyset$;

(b) *for any two atoms and $\alpha, \beta \in L$, and any rule $r \in \Pi$ and substitution θ, if $\alpha = Head(r\theta)$ and $\beta \in Body(r\theta)$, then $Body(r\theta)$ can be separated into two parts B_h and B_b, such that*:
   - $B_h \cap B_b = \emptyset$ *and* $B_h \cup B_b = Body(r\theta)$;
   - $\beta \in B_b$;
   - $Var(\{\alpha\} \cup B_h) \cap Var(B_b) = \bigcap_{\alpha' \in L} Var(\alpha')$,

then Π is a loop-separable program.

Proof. We show this by contradiction. Let Π be not a loop-separable program, and $(\alpha_1, \rho_1), \ldots, (\alpha_n, \rho_n)$ be a loop pattern of neither case 1 nor case 2 in Definition 5.

Let $\overline{y}/\overline{y'}$ be the witness of $\alpha_1 \sim \alpha_n$, and $\bar{x}$ the tuple of variables in $Var(\alpha_1) \cap Var(\alpha_n)$. We can see that $\bigcap_{i=1}^{n} Var(\alpha_i) = \bar{x}$ by noticing that we always use new variables for local variables of a rule in derivation path. Let $\theta = \overline{y}/\overline{y'} \cup \overline{y'}/\overline{y} \cup \bar{z}/\overline{z'}$ where $\bar{z}$ is the tuple of all variables in $(\alpha_1, \rho_1), \ldots, (\alpha_n, \rho_n)$ except variables in $\bar{x} \cup \overline{y} \cup \overline{y'}$, and $\overline{z'}$ a tuple of new variables.

We can see that $(\alpha_1\theta, \rho_1\theta), \ldots, (\alpha_n\theta, \rho_n\theta)$ is also a loop pattern of neither case 1 nor case 2 in Definition 5. Consider the set of atoms $L = \{\alpha_1, \ldots, \alpha_n, \alpha_1\theta, \ldots, \alpha_n\theta\}$, we can see that L is a loop by noticing that $\alpha_1\theta = \alpha_n$, and $\alpha_n\theta = \alpha_1$. We also have $\bigcap_{\alpha' \in L} Var(\alpha') = \bar{x}$, and thus $\alpha_1, \ldots, \alpha_n (= \alpha_1\theta), \ldots, \alpha_n\theta (= \alpha_1)$ is a cycle of neither case (a) nor case (b) in Proposition 18, which contradicts to the assumption of Π. □

However, there are loop-separable programs that are not covered by the conditions in Proposition 18.

Example 11. Let Π_7 be the program:

$$P(x, y) \leftarrow Q(u, v),$$
$$Q(x, y) \leftarrow P(u, v).$$

There are two loop patterns of Π_7:

$$\begin{aligned}
lp_1:\ & \big(P(x_1, x_2), P(x_1, x_2) \leftarrow Q(x_3, x_4)\big),\\
& \big(Q(x_3, x_4), Q(x_3, x_4) \leftarrow P(x_5, x_6)\big),\\
& \big(P(x_5, x_6), P(x_5, x_6) \leftarrow Q(x_7, x_8)\big),\\
lp_2:\ & \big(Q(x_1, x_2), Q(x_1, x_2) \leftarrow P(x_3, x_4)\big),\\
& \big(P(x_3, x_4), P(x_3, x_4) \leftarrow Q(x_5, x_6)\big),\\
& \big(Q(x_5, x_6), Q(x_5, x_6) \leftarrow P(x_7, x_8)\big).
\end{aligned}$$

Both lp_1 and lp_2 are loop patterns of case 1 in Definition 5, but if we consider loop

$$L = \big\{P(x_1, x_2), Q(x_1, x_2), P(x_1, x_3)\big\}$$

of program Π_7, we can see that L is neither a loop of (a) nor (b) in Proposition 18.

Example 11 also shows the reason why we need the notion loop pattern to define the loop-separable program.

6.4. Safe programs

Lee and Meng recently also identified a subclass of first-order definable programs named *safe programs* [19]. By restricting to our definition of program, a program Π is *safe* if for every rule $r \in \Pi$, every variable occurring in the rule also occurs in $Pos(r)$. However, when extensional databases are taken into account, a safe program is not necessarily first-order definable. For instance, the program Π_1 in Example 1 is a safe program, and can be proved that it is not first-order definable under our context [5].

7. Concluding remarks

In this paper, we have studied a notion of first-order definability for first-order answer set program with extensional database. Our main contribution is in identifying a non-trivial class of programs that are first-order definable on finite structures. This class, what we called loop-separable programs, is defined based on a detailed analysis of first-order loops, and contains several other interesting classes of first-order definable programs.

As we have mentioned in Section 1, study on first-order definability for answer set programs has an important application value. In another recent paper [2], we have proposed an approach to implement a first-order ASP solver, where we have shown that under finite structures, every normal logic program can be translated to a first-order sentence within a larger signature. By developing a proper first-order grounder, we can then further implement an SAT based ASP solver. What makes our results presented in this paper useful is that for loop-separable programs, the translation from the program to a first-order sentence could be much simpler.

One future work is to discover more classes of first-order definable programs, especially those that generalize our class of loop-separable programs. The notion of loops and loop formulas has been extended to disjunctive programs [18,19]. It could be interesting to consider whether our result can be extended to disjunctive programs. Another important future work is to study computational properties of loop-separable programs. We proved that the class of loop-separable programs is decidable. However, in general, deciding whether a program is loop-separable is expensive. It would be interesting to see whether a non-trivial tractable subclass can be identified.

Acknowledgements

We thank the reviewers for their valuable comments which have helped us to improve this paper.

The first author is supported in part by China NSFC 60703095 and Guangdong GDSF 07300237. The second author is supported in part by China NSFC 60573009 and 60963009 and Hongkong RGC GRF 616909. The third and fourth authors are supported in part by an Australian Research Council Discovery Grant DP0988396.

References

[1] M. Ajtai, Y. Gurevich, Datalog vs first-order logic, Journal of Computer and Systems Science 49 (1994) 562–588.
[2] V. Asuncion, F. Lin, Y. Zhang, Y. Zhou, Ordered completion for first-order logic programs on finite structures, in: Proceedings of AAAI-2010, 2010, pp. 249–254.
[3] S. Chaudhuri, M.Y. Vardi, On the equivalence of recursive and nonrecursive datalog programs, in: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on PODS, 1992, pp. 55–66.
[4] Y. Chen, F. Lin, Y. Wang, M. Zhang, First-order loop formulas for normal logic programs, in: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR-2006), 2006, pp. 298–307.
[5] Y. Chen, Y. Zhang, Y. Zhou, First-order indefinability of answer set programs on finite structures, in: Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-2010), 2010, pp. 285–290.
[6] S.S. Cosmadakis, On the first-order expressibility of recursive queries, in: Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on PODS, 1989, pp. 311–323.
[7] P. Doherty, W. Lukaszewicz, A. Szalas, Computing circumscription revisited: A reduction algorithm, Journal of Automated Reasoning 18 (1997) 297–336.
[8] P.M. Dung, K. Kanchanasut, On the generalized predicate completion of non-Horn programs, in: Proceedings of NACLP'89, 1989, pp. 604–625.
[9] S. Dworschak, S. Grell, V.J. Nikiforova, T. Schaub, J. Selbig, Modeling biological networks by action languages via answer set programming, Constraints 13 (2008) 21–65.
[10] T. Eiter, J. Lu, V.S. Subrahmanian, Computing non-ground representations of stable models, in: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-1997), 1997, pp. 198–217.
[11] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, H. Tompits, Combining answer set programming with description logics for the semantic web, Artificial Intelligence 172 (2008) 1495–1539.
[12] E. Erdem, O. Erdem, F. Türe, HAPLO-ASP: Haplotype inference using answer set programming, in: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-2009), 2009, pp. 573–578.
[13] P. Ferraris, J. Lee, V. Lifschitz, A new perspective on stable models, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007), 2007, pp. 372–379.
[14] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set solving, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007), 2007, pp. 386–392.
[15] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the 5th International Conference and Symposium on Logic Programming, 1988, pp. 1070–1080.
[16] G. Gottlob, S. Marcus, A. Nerode, G. Salzer, V.S. Subrahmanian, A non-ground realization of the stable and well-founded semantics, Theoretical Computer Science 166 (1996) 221–262.
[17] P.G. Kolaitis, Implicit definability on finite structures and unambiguous computations (preliminary report), in: Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90), 1990, pp. 168–180.
[18] J. Lee, V. Lifschitz, Loop formulas for disjunctive logic programs, in: Proceedings of the 19th International Conference on Logic Programming (ICLP-2003), 2003, pp. 451–465.
[19] J. Lee, Y. Meng, On loop formulas with variables, in: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR-2008), 2008, pp. 444–453.
[20] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (2006) 499–562.
[21] F. Lin, A study of nonmonotonic reasoning, PhD thesis, Stanford University, 1991.
[22] F. Lin, Y. Zhao, ASSAT: Computing answer sets of a logic program by SAT solvers, Artificial Intelligence 157 (2004) 115–137.
[23] F. Lin, Y. Zhou, From answer set logic programming to circumscription via logic of GK, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007), 2007, pp. 441–446.
[24] L. Löwenheim, Über Möglichkeiten im Relativekalkül, Mathematische Annalen (1915) 137–148.
[25] D. Pearce, A. Valverde, Quantified equilibrium logic and foundations for answer set programs, in: Proceedings of the 24th International Conference on Logic Programming (ICLP-2008), 2008, pp. 546–560.
[26] N. Pelov, M. Denecker, M. Bruynooghe, Well-founded and stable semantics of logic programs with aggregates, Theory and Practice of Logic Programming 7 (2007) 301–353.
[27] A. Polleres, From SPARQL to rules (and back), in: Proceedings of the 16th International Conference on World Wide Web (WWW-2007), 2007, pp. 787–796.
[28] T. Syrjänen, I. Niemelä, The smodels system, in: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-2001), 2001, pp. 434–438.
[29] N. Tran, C. Baral, Hypothesizing about signaling networks, Journal of Applied Logic 7 (2009) 253–274.
[30] M.H. van Emden, R.A. Kowalski, The semantics of predicate logic as a programming language, Journal of ACM 23 (1976) 733–742.