Università degli Studi della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXII Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

Normal Form Nested Programs

Annamaria Bria

Supervisori

Prof. Wolfgang Faber
Prof. Nicola Leone

Coordinatore

Prof. Nicola Leone

Anno Accademico 2008 - 2009

Normal Form Nested Programs

Annamaria Bria

Sommario

Nella Programmazione Logica Disgiuntiva (*PLD*) le regole sono costituite da una testa e da un corpo. La testa è una disgiunzione di atomi, mentre il corpo una congiunzione di letterali. La *PLD*, sotto la semantica degli Answer Set [22, 12] è ampiamente riconosciuta come importante strumento per la rappresentazione della conoscenza e del ragionamento non monotono.

Lifschitz, Tang e Turner [18] hanno esteso la semantica degli Answer Set (solo nel caso proposizionale o ground) ad una classe di programmi logici dove la testa e il corpo delle regole contengono espressioni innestate. Per espressioni innestate si intendono congiunzioni, disgiunzioni e negazione innestati arbitrariamente. Questa nuova classe di programmi è chiamata *Programmi Logici Innestati* e generalizza la classe dei programmi logici disgiuntivi proposizionali. Inoltre, i programmi logici innestati possono essere trasformati in programmi logici disgiuntivi, come mostrato da Lifschitz, Tang e Turner in [18] e da Pearce, et al. in [20]. Questi risultati ci permettono di valutare i programmi logici innestati usando i sistemi esistenti che supportano la *DLP*, come DLV [16], GnT [13], oppure Cmodels3 [17]. Si noti che le trasformazioni che consentono di ottenere i programma *PLD* introducono nuovi simboli e pertanto il programma risultante non è equivalente al programma originario nel senso classico. Ad ogni modo esiste una corrispondenza biunivoca tra gli answer set del programma innestato con gli answer set del programma trasformato. Sfortunatamente queste trasformazioni funzionano solo per programmi proposizionali e dunque non si possono usare le variabili, uno dei punti di forza dei programmi logici disgiuntivi. Questa limitazione diminuisce drasticamente l'applicabilità dei programmi logici innestati soprattutto quando il ragionamento è fatto su un numero molto grande di fatti.

Una generalizzazione di queste tecniche a programmi che contengono variabili, non è ovvia. Per esempio, aggiungendo semplicemente le variabili al

metodo descritto in [20] si ottengono regole dipendenti dal dominio. Intuitivamente un programma che contiene variabili è dipendente dal dominio se la semantica dipende dal particolare dominio che viene scelto per la sua interpretazione. Questa proprietà è stata studiata per la prima volta nel contesto dei sistemi di basi di dati (vedi [1] per approfondimenti). Per i programmi *DLP* l'indipendenza dal dominio è assicurata imponendo ai programmi delle condizioni sintattiche. Una di queste condizioni è nota come safety e per le regole *PLD* significa che ogni variabile che compare nella regola deve comparire anche in almeno un letterale positivo del corpo.

Motivati da queste considerazioni, in questo lavoro di tesi estendiamo i programmi logici disgiuntivi non-ground ad una classe di programmi nei quali la testa delle regole è una formula in forma normale disgiuntiva composta da atomi, mentre il corpo è una formula in forma normale congiuntiva composta da letterali. Questi programmi sono chiamati programmi Normal Form Nested (*NFN*) e diversamente dai programmi logici innestati di [18] possono contenere variabili. In questo lavoro di tesi abbiamo studiato la semantica e la proprietà di indipendenza dal dominio e una trasformazione polinomiale dai programmi *NFN* ai programmi *PLD*, che preserva la safety.

Il bisogno di estendere la *PLD* con congiunzione nella testa e disgiunzione nel corpo deriva molto spesso da problemi del mondo reale. Ad esempio, il seguente problema si incontra spesso nelle applicazioni di data-integration.

Sia $p(ID, nome, cognome, anni)$ una relazione globale (per persone) con un vincolo di chiave sul primo attributo ID. Per realizzare un corretto query-answering se due tuple condividono la stessa chiave, la relazione persona viene "riparata" cancellando intenzionalmente una delle due tuple. In *PLD*, questo problema viene modellato dalle seguenti regole (dove $\overline{p}$ rappresenta la tupla da cancellare e p' la relazione consistente risultante sulla quale vengono calcolate le query).

$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), N \neq M.$$
$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), S \neq T.$$
$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), A \neq B.$$
$$p'(I,N,S,A) \text{ :- } p(I,N,S,A), \textbf{not } \overline{p}(I,N,S,A).$$

La prima regola indica che una delle due tuple viene cancellata se esse condividono la stessa chiave ma hanno nomi diversi. Analogamente, la seconda cancella una delle due tuple se esse hanno la stessa chiave ID ma cognomi diversi. La terza cancella una delle due tuple se esse hanno la stessa chiave ma anni diversi. L'ultima regola definisce la tabella riparata.

Le prime tre regole *DLP* possono essere equivalentemente rappresentate da una sola regola *NFN*, la quale è più succinta e quindi più leggibile:

$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), (N \neq M \vee S \neq T \vee A \neq B).$$

In particolare, la regola *NFN* rappresenta la cancellazione di una delle due tuple se queste hanno lo stesso ID e nomi diversi, oppure cognomi diversi oppure anni diversi.

Come ulteriore esempio, consideriamo un problema dal mondo della *toeria dei grafi*: co-CERT3COL—che generalizza la 3-uncolorability, dovuto a I. Stewart [24]. Dato un grafo G, i cui archi sono etichettati con un insieme non vuoto di variabili $v_1, \ldots, v_n$, trovare un assegnamento di verità per $v_1, \ldots, v_n$ tale che il sottografo G' di G, contenente tutti gli archi e tale che almeno un letterale nelle etichette di e sia vero, non è 3-colorabile.

Supponiamo che gli archi etichettati siano rappresentati dai predicati $p(X,Y,V)$, per indicare che l'arco che connette X ed Y ha un'etichetta positiva, e $n(X,Y,V)$ per indicare che l'arco che connette X ed Y ha un'etichetta negativa V.

Nella seguente tabella, sulla sinistra riportiamo il programma *DLP* definito in [5] che risolve co-CERT3COL, mentre sulla destra riportiamo un programma *NFN* equivalente.

DLP encoding	*NFN encoding*
r_1 : $v(X)$:- $p(X,Y,V)$. r_2 : $v(Y)$:- $p(X,Y,V)$. r_3 : $v(X)$:- $n(X,Y,V)$. r_4 : $v(Y)$:- $n(X,Y,V)$.	r_a : $v(X), v(Y)$:- $p(X,Y,V) \vee n(X,Y,V)$.
r_5 : $bool(V)$:- $p(X,Y,V)$. r_6 : $bool(V)$:- $n(X,Y,V)$. r_7 : $t(V) \vee f(V)$:- $bool(V)$.	r_b : $t(V) \vee f(V)$:- $p(X,Y,V) \vee n(X,Y,V)$.
r_8 : $c(X,r)$:- $w, v(X)$. r_9 : $c(X,g)$:- $w, v(X)$. r_{10} : $c(X,b)$:- $w, v(X)$.	r_c : $c(X,r), c(X,g), c(X,b)$:- $w, v(X)$.
r_{11} : $c(X,r) \vee c(X,g) \vee c(X,b)$:- $v(X)$. r_{12} : w :- $p(X,Y,V), t(V), c(X,A), c(Y,A)$. r_{13} : w :- $n(X,Y,V), f(V), c(X,A), c(Y,A)$.	

La regola r_a sostituisce le regole da r_1 a r_4, la regola r_b sostituisce le regole da r_5 a r_7, ed il predicato *bool* non è necessario. In fine, la regola r_d sostituisce le regole da r_8 ad r_{10}. Le regole r_{11}, r_{12} e r_{13} esistono in entrambe le codifiche.

Si noti che nel linguaggio *NFN* riusciamo a rappresentare il problema con un programma molto più succinto ed inoltre non abbiamo bisogno del predicato intermedio *bool*.

Contributi I principali contributi della tesi possono essere riassunti come segue.

1. Abbiamo esteso la *DLP* introducendo la congiunzione nella testa delle regole e la disgiunzione nel corpo delle stesse, ottenendo una nuova classe di programmi: *i programmi NFN*.

2. Abbiamo studiato le proprietà dei programmi *NFN* mostrando i seguenti risultati.

   - Abbiamo dato la definizione di *Safety* per i programmi *NFN* e dimostrato che ogni programma safe è indipendente dal dominio (cioè esso ha gli stessi answer set su ogni universo che estende le costanti del programma).
   - Gli answer set per i programmi *NFN* coincidono con gli answer set definiti da Lifschitz et al. in [18] per i programmi *NLP*, sul segmento del linguaggio comune.
   - Gli answer set per i programmi *NFN* coincidono con i modelli stabili di Herbrand definiti da Ferraris et al. in [7] per le formule che corrispondono ai programmi *NFN*.
   - Gli answer set per i programmi *NFN* coincidono con gli answer set per i programmi *DLP* definiti da Gelfond e Lifschitz in [12].
   - La nostra definizione di Safety per i programmi *NFN* è più generale di quella definita da Lee et al. in [15] sui programmi *NFN*, nel senso che ci sono programmi che non sono safe nella definizione in [15] ma che sono safe secondo la nostra definizione.

3. Abbiamo progettato un algoritmo che trasforma i programmi *NFN* in programmi *DLP* in modo efficiente e abbiamo dimostrato che esso soddisfa importanti proprietà.

- La trasformazione preserva la safety.
- La taglia del programma trasformato è polinomiale nella taglia del programma *NFN*.
- Esiste una corrispondenza biunivoca tra gli answer set del programma *NFN* e quelli del programma riscritto. In questo modo si possono ottenere gli answer set del programma *NFN*, da quelli del programma trasformato, con una semplice proiezione.

4. Abbiamo realizzato un sistema, `nfn2dlp`, che implementa l'algoritmo di riscrittura. Inoltre abbiamo implementato anche un sistema che calcola gli answer set di un programma *NFN*: `nfnsolve`. Entrambi i tool sono disponibili al sito `http://www.mat.unical.it/software/nfn2dlp/`.

Contents

List of Figures

Introduction

Context and Motivation

In Disjunctive Logic Programming (*DLP*) the heads (resp. the bodies) of rules are disjunctions (resp. conjunctions) of simple constructs, viz. atoms and literals. *DLP*, under the answer set semantics [22, 12], is established as an important tool for knowledge representation and reasoning [2].

Lifschitz, Tang and Turner [18] extended the answer set semantics (in the propositional or ground case) to a class of logic programs where the heads and the bodies of rules are nested expressions. These expressions are formed from negation-as-failure literals, conjunction and disjunction, nested arbitrarily. This class of programs, called *nested logic programs*, generalizes the class of (ground) disjunctive logic programs. Moreover, as shown in [18, 20], nested logic programs can be transformed into disjunctive logic programs. These results allow for evaluating ground nested logic programs using *DLP* systems, such as DLV [16], GnT [13], or Cmodels3 [17]. Note that these methods introduce new symbols implying that the result is not equivalent in the classical sense to the original program. Anyway, there is a one-to-one correspondence between the answer sets. However, given that these transformations work only for ground nested logic programs, one of the strongest features of logic programming, namely variables, cannot be used in problem representations. This restriction limits the suitability of nested logic programs in many application domains, especially when reasoning is to be done on large numbers of input facts.

Unfortunately, a generalization of these techniques to programs with variables is not straightforward. A major obstacle is domain dependence, a property first studied in the realm of database systems (cf. [1] for a summary). Essentially, when variables are present, the semantics of rules will in general depend on the particular domain that is chosen for their interpretation.

This entails several undesirable effects such as a strong dependence on the context, even if this context is completely independent, issues with finiteness and in general unintuitive semantics. When one would just add variables to the method of [20], one easily obtains domain dependent rules.

Domain dependence is also an issue in *DLP*, and in this context (as in databases) a syntactic requirement is imposed on programs, which guarantees domain independence and therefore avoids all of the problems that domain dependence entails. This requirement is known as safety, which for *DLP* rules means that each variable in a rule must occur in a positive body literal.

Motivated by these considerations, we extend non-ground *DLP* to a class of programs, in which rule heads are formulas in disjunctive normal form consisting of atoms, and in which the rule bodies are formulas in conjunctive normal form consisting of literals. These programs are referred to as Normal Form Nested (*NFN*) programs, and are different to nested logic programs of [18], since they may contain variables. We study semantic and domain dependence properties of this class of programs, and provide a definition of safety (which guarantees domain independence) and a polynomial translation from *NFN* programs to *DLP*, which maintains safety.

The need for extending *DLP* with conjunction in the heads and disjunction in the body arises quite often in real world applications.

Databases

In this section we show an example that we met in a real-world data-integration application.

Consider a global relation $p(ID, name, surname, age)$ (for persons) with a key-constraint on the first attribute ID. To perform consistent query answering [3], when two tuples share the same key, the relation person is "repaired" by intensionally deleting one of them. In *DLP*, this is obtained by the following rules (where $\overline{p}$ stands for deleted tuples, and p' is the resulting consistent relation on which query answers are computed).

$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), N \neq M.$$
$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), S \neq T.$$
$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), A \neq B.$$
$$p'(I,N,S,A) \text{ :- } p(I,N,S,A), \textbf{not } \overline{p}(I,N,S,A).$$

The first rule deletes one of two tuples sharing the same key and having different names. Similarly, the second rule deletes one of two tuples sharing the same key and having different surnames. Finally, the third rule deletes

one of two tuples if they have the same ID but different ages. The last rule builds the repaired database.

The first three *DLP* rules can be equivalently encoded by a single *NFN* rule, which is much more succinct and readable:

$$\overline{p}(I,N,S,A) \vee \overline{p}(I,M,T,B) \text{ :- } p(I,N,S,A), p(I,M,T,B), \\ (N \neq M \vee S \neq T \vee A \neq B).$$

In detail, the *NFN* rule deletes one of two tuples if the tuples have the same ID and different names, different surnames or different ages.

Graph Theory

For a more involved example, we consider a problem from graph theory. In particular, we consider the problem co-CERT3COL – a generalization of graph 3-uncolorability, due to I. Stewart [24]. Given a graph G, whose edges are labeled with nonempty sets of variables $v_1, \ldots, v_n$, find a truth assignment to $v_1, \ldots, v_n$ such that the subgraph G' of G, containing all edges e such that at least one literal in the label of e is satisfied, is not 3-colorable.

Let the labeled edges of G be represented by predicates $p(X,Y,V)$ and $n(X,Y,V)$, indicating that the edge connecting vertices X and Y has a positive or negative label V, respectively.

On the left of the following table we report a *DLP* encoding as defined in [5], whereas on the right we report an equivalent *NFN* encoding. Rules r_{11} to r_{13} belong to both encodings. In the *NFN* version we save seven rules and the intermediate predicate *bool*.

DLP encoding	*NFN encoding*
$r_1:\ v(X) \text{ :- } p(X,Y,V).$ $r_2:\ v(Y) \text{ :- } p(X,Y,V).$ $r_3:\ v(X) \text{ :- } n(X,Y,V).$ $r_4:\ v(Y) \text{ :- } n(X,Y,V).$	$r_a:\ v(X), v(Y) \text{ :- } p(X,Y,V) \vee n(X,Y,V).$
$r_5:\ bool(V) \text{ :- } p(X,Y,V).$ $r_6:\ bool(V) \text{ :- } n(X,Y,V).$ $r_7:\ t(V) \vee f(V) \text{ :- } bool(V).$	$r_b:\ t(V) \vee f(V) \text{ :- } p(X,Y,V) \vee n(X,Y,V).$
$r_8:\ c(X,r) \text{ :- } w, v(X).$ $r_9:\ c(X,g) \text{ :- } w, v(X).$ $r_{10}:\ c(X,b) \text{ :- } w, v(X).$	$r_c:\ c(X,r), c(X,g), c(X,b) \text{ :- } w, v(X).$
$r_{11}:\ c(X,r) \vee c(X,g) \vee c(X,b) \text{ :- } v(X).$ $r_{12}:\ w \text{ :- } p(X,Y,V), t(V), c(X,A), c(Y,A).$ $r_{13}:\ w \text{ :- } n(X,Y,V), f(V), c(X,A), c(Y,A).$	

Here, rule r_a replaces r_1 to r_4, rule r_b replaces r_5 to r_7, and the intermediate predicate *bool* is not needed. Finally, rule r_d replaces rules r_8 to r_{10}.

Contribution

The main contributions are the following:

- We extend *DLP* with variables introducing conjunctions in the head of rules and disjunctions in the body of rules, obtaining a new language, *NFN* programs. We formally define the syntax and semantics of this language, based on a new notion of answer set.
- We study the properties of *NFN* programs showing the following results:
  - We provide a definition of safe *NFN* programs. We show that every safe program is domain independent, that is, it has the same answer sets on each universe extending the constants of the program.
  - The answer sets for *NFN* programs coincide with the answer sets defined by Lifschitz, Tang, and Turner in [18] for *NLP* programs, on the common language fragment.
  - The answer sets for *NFN* programs coincide with the Herbrand stable models defined by Ferraris, Lee, and Lifschitz in [7] for formulas that correspond to *NFN* programs.
  - The answer sets for *NFN* coincide with the standard answer sets defined by Gelfond and Lifschitz in [12] on (possibly non ground) *DLP* programs.
  - Our definition of safety is more general than the one defined by Lee, Lifschitz, and Palla in [15] on *NFN* programs, in the sense that there are programs that are unsafe in the definition of [15], but safe in our definition.
- We present an algorithm that transforms normal form nested programs to disjunctive logic programs and show that it satisfies several important properties:

  - The transformation preserves safety.
  - The size of the transformed program is polynomial in the size of the original *NFN* program.
  - There is a one-to-one correspondence between the answer sets of the original and the transformed program, such that the answer sets of the original program can be read off the answer sets of the transformed programs by a simple projection.

- We provide:
  - A tool implementing the presented efficient translation from safe *NFN* programs to safe *DLP* programs, called `nfn2dlp`.
  - The tool `nfnsolve` that computes answer sets for *NFN* programs.
  - At `http://www.mat.unical.it/software/nfn2dlp/` both tools are publicly available.

Structure of the Thesis

The thesis is organized as follows.

- Chapter 1 reviews Disjunctive Logic Programs. Syntax and semantics of the underlying disjunctive logic language are reported. Then, an important issue is discussed: Domain Independence, the question of whether the semantics is independent of the considered domain. Finally the class of safe programs is described that are guaranteed to be domain independent.

- Chapter 2 introduces the *NFN* programs, an extension of disjunctive logic programs with variables. Syntax and semantics are formally defined. Moreover, the most important properties of the *NFN* programs are studied.

- Chapter 3 describes the Guess and Check (**G&C**) technique for Knowledge Representation and Reasoning using *NFN* programs. Furthermore, it is illustrated how the **G&C** technique can be applied on a number of examples.

- Chapter 4 describes a naïve transformation of *NFN* programs to *DLP* programs based on distributivity that is in general not polynomial . Then, an efficient algorithm transforming *NFN* to *DLP* programs is provided that introduces new symbols. Finally, the computational properties of the algorithm are studied.
- Chapter 5 presents the tool `nfn2dlp`, a compiler for *NFN* programs. Moreover, a solver for *NFN* programs, called `nfnsolve`, is presented. The implementation and the usage of these tools are described in detail.
- Chapter 6 related works are discussed.

Chapter 1

Disjunctive Logic Programs
DLP

In this chapter we present the *Disjunctive Logic Programming* (*DLP*) framework, that has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. In particular, we define the syntax of the underlying disjunctive logic language and we describe the associated answer set semantics. Since with the presence of variables, domain independence is no longer guaranteed, we describe the class of the *DLP* safe programs, which are guaranteed to be domain independent.

The chapter is organized as follows:

- ★ Sections 1.1 and 1.2 provide a formal definition of the syntax and semantics of disjunctive logic programs.
- ★ In Section 1.3 we give the definition of domain independence and safety.

1.1 Syntax

Let $\mathcal{A}$ be the alphabet containing digits, letters and "_". We call finite sequences of symbols from $\mathcal{A}$ *strings* or *words* over $\mathcal{A}$. $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$.

Definition 1.1 (Predicates)**.** *The set* Π *of predicates is the set of all string beginning with a lower-case letter.*

Examples of predicates in Π are p, $person$ and $delete$.

Definition 1.2 (Constants)**.** *The set* C *of constant symbols is the set of all strings* $s \in \mathcal{A}^*$ *such that* s *begins with a lower-case letter or digit.*

Examples of constants in C are x, $5A\_b$ and 123.

Definition 1.3 (Variables)**.** *The set* V *of variable symbols is the set of all strings* $s \in \mathcal{A}^*$ *such that* s *begins with an uppercase letter.*

For example, the strings X, $V2f$ and Vi_X3 are variables.

Definition 1.4 (Term)**.** *A term is a constant in* C *(see Definition 1.2) or a variable in* V *(see Definition 1.3).*

Definition 1.5 (Atom)**.** *Let* $p \in \Pi$ *be a predicate and* $t_1, \ldots, t_n$ *terms,* $n \geq 0$*; then an atom of arity* n *is in the form* $p(t_1, \ldots, t_n)$*. If* $n = 0$ *the parentheses around terms are omitted.*

For example $node(X)$, $a(V2f, 123)$ and $true$ are atoms.

Definition 1.6 (Literal)**.** *A literal is an atom* a *(*positive literal*) or a negated atom* ***not*** a *(*negative literal*).*

We are now able to give the definition of *DLP* rules.

Definition 1.7 (Disjunctive Logic Rule)**.** *A* (disjunctive logic) rule r *is a formula*

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \text{:-} \ b_1, \ldots, b_k, \ \textbf{\textit{not}} \ \ b_{k+1}, \ldots, \ \textbf{\textit{not}} \ \ b_m. \tag{1.1}$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ *are atoms and* $n \geq 0$, $m \geq 0$ *and* $k \geq 0$.

The disjunction $a_1 \vee \cdots \vee a_n$ is called *head* of r, while the conjunction $b_1, \ldots, b_k$, **not** $b_{k+1}, \ldots,$ **not** b_m is the *body* of r. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), the rule is called *fact* and we usually omit the :- sign.

Example 1.1 (*DLP* rule)**.** *The following is a* (disjunctive logic) rule*:*

$$male(X) \vee female(X) \text{ :- } person(X). \tag{1.2}$$

The rule stands for "if X is a person then X is male of female" *. The head of the rule is the disjunction* $male(X) \vee female(X)$ *while the body is* $person(X)$*. Since the body rule contains only positive literals, the rule is positive.*

An *(integrity) constraint* is a rule such that $H(r) = \emptyset$ (i.e. $n = 0$)

$$\text{:- } b_1, \ldots, b_k, \textbf{ not } b_{k+1}, \ldots, \textbf{ not } b_m. \tag{1.3}$$

Definition 1.8 (*DLP* program)**.** *A* disjunctive logic program *(DLP program)* $\mathcal{P}$ *is a finite set of rules, possibly including integrity constraints.*

If all rules in $\mathcal{P}$ are positive than $\mathcal{P}$ is called *positive*. If all rules are normal then $\mathcal{P}$ is called *normal logic program*.

Under the Answer Set Semantics, a constraint

$$\text{:- } body.$$

can be simulated through the introduction of a standard rule:

$$fail \text{ :- } body, \textbf{not } fail.$$

where $fail$ is a fresh predicate not occurring elsewhere in the program. So, from a semantic point of view, we can assume that there are no constraints.

1.2 Semantics

The most widely accepted semantics for *DLP* programs is based on the notion of answer set, proposed in [12] as a generalization of the concept of stable model [11].

1.2.1 Program Instantiation

The use of variables in a program allows for obtaining a more compact representation of problems. In particular, variables are an abstraction of constants.

Example 1.2. *Let us consider the following program P:*

$a(X)$:- $b(X)$.
$b(1)$.
$b(2)$.

P is a shorthand for the program P_g which is the instantiation of P:

$a(1)$:- $b(1)$.
$a(2)$:- $b(2)$.
$b(1)$
$b(2)$.

Let Σ be a symbol denoting a general expression (literal, disjunct, conjunct, etc.). In the following, we denote by $const(\Sigma)$ the set of constants that appear in a construct Σ and by $vars(\Sigma)$ the set of variables that appear in Σ.

Definition 1.9 (Universe and Base of Herbrand)**.** *Let P a program then*

- *the* Herbrand Universe (Domain) *U_P of P, is the set of constants appearing in P;*
- *the* Herbrand Base *B_P of P, is the set of all ground atoms obtainable from the atoms of P by replacing variables with elements from U_P.*

In case no constant appears in P an arbitrary constant c is added to U_P.

Example 1.3. *Let us consider the program P_1 of the Example 1.2, then*

$$U_{P_1} = \{1, 2\}$$

and

$$B_{P_1} = \{a(1), a(2), b(1), b(2)\}.$$

Definition 1.10 (Substitution)**.** *Let $U \supseteq U_P$ be a set of constants and r a DLP rule. A* substitution *is total function $\sigma : vars(r) \mapsto U$ that maps each variable of r to a constant in U.*

Next we denote a substitution σ also as the set $\{X/c \mid \sigma(X) = c\}$.

Definition 1.11 (Ground Instances)**.** *Given a substitution σ, a ground instance of a construct Σ w.r.t. σ is the application of σ on variables of Σ, denoted by $\Sigma\sigma$.*

A construct without variables is called *ground*.

Example 1.4. *Let us consider the rule*

$$r = a(X) \text{ :- } b(X)$$

of the Example 1.2, $U = \{1\}$ and the substitution is $\sigma = \{X/1\}$. Then, $r\sigma$ is the ground rule

$$a(1) \text{ :- } b(1).$$

Definition 1.12 (Instantiation of a rule)**.** *The* instantiation *of a rule r, denoted by $Ground(r, U)$, is the set of ground instances of r w.r.t. U, obtained by applying all possible substitutions w.r.t. r and U.*

Example 1.5. *Let U be the set $\{1, 2\}$ and $r = a(X)$:- $b(Y)$. $Ground(r, U)$ is the following set of rules:*

$a(1)$:- $b(1)$.
$a(2)$:- $b(2)$.
$a(2)$:- $b(1)$.
$a(1)$:- $b(2)$.

Definition 1.13 (Ground Program)**.** *Let P be a DLP program, the instantiation of P w.r.t. U, denoted by $Ground(P, U)$, is the union of all instantiations of rules in P:*

$$Ground(P, U) = \bigcup_{r \in P} Ground(r, U).$$

A ground program is also called a *propositional* program. As a special case, let $Ground(P) = Ground(P, U_P)$.

1.2.2 Interpretation and Models

In this section we define an answer set for a program P.

Definition 1.14 (Interpretation)**.** *Given a ground program P,* (Herbrand) interpretation *I for P is a subset of B_P.*

In the following we define when a construct is true w.r.t. an interpretation.

Definition 1.15. *Let P be a program and I be an interpretation for P:*

1. *A ground atom a is true (resp. false) w.r.t. I if $a \in I$ (resp. $a \notin I$).*
2. *A ground negative literal **not** a is true (resp. false) w.r.t. I if $a \notin I$ (resp. $a \in I$).*

Let r be a ground rule such that $r \in Ground(P)$:

3. *The head H of r is true w.r.t. I if at least one atom of H is true w.r.t. I; otherwise H is false w.r.t. I.*
4. *The body B of r is true w.r.t. I if all literals of B are true w.r.t. I; otherwise B is false w.r.t. I.*
5. *r is satisfied w.r.t. I if the head is true w.r.t. I or the body is false w.r.t. I.*

P is satisfied w.r.t. I if all rules of $Ground(P)$ are satisfied.

In the following, we denote truth and falsity of a construct Σ with $I \vDash \Sigma$ and $I \nvDash \Sigma$ respectively. Given a rule or a program Δ, we also denote the satisfiability of Δ w.r.t. I by $I \vDash \Delta$, and unsatisfiability of Δ w.r.t. I by $I \nvDash \Delta$.

Definition 1.16 (Model)**.** *Let P be a DLP program and M an interpretation for P. M is a model for P if $M \vDash P$.*

Example 1.6. *Let P the program containing only the following rule:*

$$a \vee b \vee c.$$

and $M = \{a\}$ be an interpretation for P. M is a model for P because from Definition 1.15 item 3. *$M \vDash a \vee b \vee c$, that is M is a model for P.*

Definition 1.17 (Minimal Model)**.** *Let P be a DLP program and M be an interpretation for P. M is a minimal model for P if no model N for P exists such that $N \subsetneq M$.*

Example 1.7. *In the Example 1.6, the model M is also a minimal model for P since $\emptyset \nvDash P$.*

1.2.3 Answer Set Semantics

We will next defne the notion of answer sets for *DLP* programs. First, we provide the transformation by which the reduct of a ground program w.r.t. an interpretation is formed. Note that this definition reported in [6] is an alternative to the Gelfond-Lifschitz transformation for DLP programs that has been shown to be equivalent.

Definition 1.18 (Reduct)**.** *Given a ground DLP program P and an interpretation I, let P^I denote the transformed program obtained from P by deleting rules in which a body literal is false w.r.t. I.*

Example 1.8. *Consider the following program P:*

$$a. \qquad b. \qquad d \text{ :- } \textbf{\textit{not}}\ e, b. \qquad c \text{ :- } \textbf{\textit{not}}\ b, a. \tag{1.4}$$

If $I = \{a, b, d\}$ then P^I contains the following rules:

$$a. \qquad b. \qquad d \text{ :- } \textbf{\textit{not}}\ e, b. \tag{1.5}$$

*The last rule in the program (1.4) is deleted since the literal **not** b is false w.r.t. I.*

Definition 1.19 (Answer Set)**.** *Given a DLP program P, an interpretation I of $Ground(P)$ is an* answer set *for P if it is a subset-minimal model of $Ground(P)^I$.*

Note that any answer set I of P is also a model of P because $Ground(P)^I \subseteq Ground(P)$, and rules in $Ground(P) \setminus Ground(P)^I$ are satisfed w.r.t. I.

Example 1.9. *In Example 1.8, I is an answer set of P since I is a model for P^I and no subset of I satisfies all rules in (1.5).*

For a more involved example, let us consider the following program Q :

$a(X) \vee b(Y)$:- $c(X, Y), \textbf{\textit{not}}\ d(X).$
$d(1).$
$c(1, 2).$
$c(2, 1).$

The instantiation of Q, $Ground(Q)$, is the following ground program:

$a(1) \vee b(1)$:- $c(1, 1), \textbf{\textit{not}}\ d(1).$
$a(2) \vee b(2)$:- $c(2, 2), \textbf{\textit{not}}\ d(2).$
$a(1) \vee b(2)$:- $c(1, 2), \textbf{\textit{not}}\ d(1).$
$a(2) \vee b(1)$:- $c(2, 1), \textbf{\textit{not}}\ d(2).$
$c(1, 2).$
$c(2, 1).$
$d(1).$

If we consider the interpretation $J = \{d(1), c(1,2), c(2,1), a(2)\}$, $Ground(Q)^J$ *is the following program:*

$a(2) \vee b(1)$:- $c(2,1)$, ***not*** $d(2)$.
$c(2,1)$
$c(1,2)$.
$d(1)$.

J satisfyes all rules in $Ground(Q)^J$. *Moreover, no subset of J is a model for* $Ground(Q)^J$, *therefore J is an answer set of Q.*

In the following, the set of answer sets for P is denoted by $AS(P)$.

1.3 Domain Independence and Safety

In this section we study the property of domain independence and define a syntactic criterion for identifying *DLP* programs which are guaranteed to be domain independent.

1.3.1 Domain Independence

Let us first examine some examples that highlight the importance of domain independence, which will be formally defined afterwards.

Example 1.10. *Let consider the program* P_u

$a(X)$:- $b(Y)$.
$b(1)$.

where $U_{P_u} = \{1\}$ *then:* $Ground(P_u, U_{P_u})$ *is the following:*

$a(1)$:- $b(1)$.
$b(1)$.

It is easy to see that $I = \{a(1), b(1)\}$ *is a minimal model for* $Ground(P_u, U_{P_u})^I$, *and no other interpretation has this property, therefore* $AS(P_u, U_{P_u}) = \{I\}$.

If we consider as universe $U = \{1, 2\}$, $Ground(P_u, U)$ *is the following program:*

$b(1)$.
$a(1)$:- $b(1)$.
$a(1)$:- $b(2)$.
$a(2)$:- $b(1)$.
$a(2)$:- $b(2)$.

The interpretation $J = \{a(2), a(1), b(1)\}$ *is a minimal model for* $Ground(P_u, U)^J$, $AS(P_u, U) = \{J\}$. *Therefore, we have different answer sets depending on the considered universe.*

As another example we consider a program containg negative literals.

Example 1.11. *Let consider the following program* P_{u2}

$a(X)$:- ***not*** $b(X)$.
$b(1)$.

$Ground(P_{u2}, \{1\})$ *is*

$a(1)$:- ***not*** $b(1)$.
$b(1)$.

The interpretation $I_2 = \{b(1)\}$ *is the only answer set for* P_{u2}. *Let us consider* $U_2 = \{1, 2\}$, *then* $Ground(P_{u2}, U_2)$ *is the following program:*

$a(1)$:- ***not*** $b(1)$.
$a(2)$:- ***not*** $b(2)$.
$b(1)$.

It is simple to see that the interpretation $J_2 = \{a(2), b(1)\}$ *is the only answer set of* P_{u2} *w.r.t.* U_2.

In the previous examples the answer sets of both programs P_u and P_{u2} depend on the considered universe.

In the following, we show an example where the answer sets does not change when the domain chages.

Example 1.12. *Let us consider the program* $\mathcal{P}$

$a(X)$:- $b(X)$.
$b(1)$.

where $U_{\mathcal{P}} = \{1\}$ *then* $Ground(\mathcal{P}, U_{\mathcal{P}})$ *is*

$a(1)$:- $b(1)$.
$b(1)$.

It is easy to see that $I = \{a(1), b(1)\}$ *is a minimal model for* $Ground(\mathcal{P}, U_{\mathcal{P}})^I$, *therefore* $AS(\mathcal{P}) = \{I\}$.

If we consider as universe $U = \{1, 2\}$*,* $Ground(\mathcal{P}, U)$ *is the following program:*

$a(1)$:- $b(1)$.
$a(2)$:- $b(2)$.
$b(1)$.

I *is also a minimal model for* $Ground(\mathcal{P}, U)^I$ *and the only answer set w.r.t.* U*. Therefore, the answer sets of* $\mathcal{P}$ *do not depend on the considered universe.*

We now formally define domain independence, stating that the semantics should be independent of the universe, as long as it is sufficiently large.

Definition 1.20 (Domain Independence)**.** *Let* P *be a DLP program and* U_P *be the set of constants appearing in* P*.* P *is* domain independent *if for each* $U \supseteq U_P$*,*

$$AS(Ground(P, U)) = AS(Ground(P, U_P))$$

holds.

Domain dependent programs have obvious weaknesses, such as those outlined in Examples 1.10 and 1.11. But the consequences of domain dependence becomes even more drastic when finite programs with infinite domains (such as domains representing numbers) are considered: While domain independent programs guarantee finite answer sets, domain dependent programs can yield infinite answer sets or an infinite number of them. For instance P_u of Example 1.11 with an infinite universe $U = U_{P_u} \cup \mathbb{N}$ will yield infinite answer sets. For further discussions on the significance of domain independence, we refer to [1].

Examples 1.10 and 1.11 directly serve as a proof for the following fact.

Fact 1.3.1. *A DLP program* P *is in general not domain independent.*

1.3.2 Safe *DLP* Programs

In this section, we recall the class of *DLP* programs, which is characterized syntactically, such that all programs in this class are guaranteed to be domain independent.

Definition 1.21 (Safe rules and programs)**.** *A DLP rule* r *is safe if each variable of* r *appears in a positive literal of the body of* r*, otherwise it is unsafe. A DLP program* P *is safe if each rule is safe, otherwise it is unsafe.*

Example 1.13. *Let us consider the DLP rule*

$$a(X) \vee b(Y) \text{ :- } a(X), \textbf{\textit{not}}\ b(X), c(Y).$$

The rule is safe indeed variable X appears in the positive literal $a(X)$ of the body. Similarly, variable Y is safe because it appears in the positive body literal $c(Y)$. The rule

$$a(X) \vee b \text{ :- } \textbf{\textit{not}}\ b(X).$$

is unsafe because the variable X does not appear in any positive body literal (but only in the head and the negative body literal). The rule

$$a(X) \text{ :- } b(Y).$$

of Example 1.10 is unsafe because variable X does not appear in any positive body literal (but only in the head). The rule

$$a(X) \text{ :- } \textbf{\textit{not}}\ b(X).$$

of Example 1.11 is unsafe. Indeed variable X appears only in the head and the negative body literal of the rule.

Theorem 1.3.1. *Let $\mathcal{P}$ a DLP program, if $\mathcal{P}$ is safe then $\mathcal{P}$ is domain independent.*

In the Section 2.3.2, we proove the Theorem 2.3.3 that is more general than the Theorem 1.3.1.

Chapter 2

Normal Form Nested Programs *NFN*

In this chapter we present the Normal From Nested (*NFN*) language. It is essentially an extension of *DLP* with variables by allowing conjunctions of atoms in place of atoms in rule heads and disjunctions of literals in place of literals in rule bodies. In the following, we formally define the syntax of the new language and the associated answer set semantics. However, with the introduction of variables an important issue arises: domain independence, the question of whether the semantics of a program is independent of the considered domain (given that it is sufficiently rich). As seen in Section 1.3, domain independence is desirable because it guarantees that the semantics remains equal if unrelated information is added and also ensures finiteness of intended models even if infinite domains are considered. In the presence of variables, *NFN* programs in general are not domain independent. We study this issue in depth and define the class of safe *NFN* programs, which are guaranteed to be domain independent.

The chapter is organized as follows:

- ★ In Sections 2.1 and 2.2 syntax and semantics of *NFN* programs are formally defined.
- ★ In Section 2.3 the most important properties of *NFN* programs are studied.

2.1 Syntax

Let $\mathcal{A}$ the alphabet containing digits, letters and "_". We call finite sequences of symbols from $\mathcal{A}$ *strings* or *words* over $\mathcal{A}$. $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$.

Variables, constants, terms, atoms and literals are defined as in Section 1.1.

Definition 2.1 (Basic Disjunction)**.** *A basic disjunction is of the form* $(k_1 \vee \cdots \vee k_n)$ *where each* $k_1, \ldots, k_n$ *is a literal; if each* $k_1, \ldots, k_n$ *is an atom, the basic disjunction is positive.*

For example $a(X) \vee b$ and $c(X)$ are basic disjunctions and since $a(X), b$ and $c(X)$ are atoms, both are positive.

Definition 2.2 (Basic Conjunction)**.** *A basic conjunction is of the form* $(l_1, \ldots, l_n)$ *where each* $l_1, \ldots, l_n$ *is a literal. If each* $l_1, \ldots, l_n$ *is an atom, the basic conjunction is positive.*

For example $b(X), d(X), c$ and c are basic conjunctions where $b(X), d(X)$ and c are literals.

The parentheses around basic conjunctions and disjunctions may be omitted in unambiguous occurrences.

Definition 2.3 (*NFN* Rule)**.** *A (normal form nested) rule* r *is of the following form:*

$$C_1 \vee \cdots \vee C_n \text{ :- } D_1, \ldots, D_m. \qquad n, m \geq 0$$

where $C_1, \ldots, C_n$ *are positive basic conjunctions and* $D_1, \ldots, D_m$ *are basic disjunctions.*

The disjunction $C_1 \vee \cdots \vee C_n$ is the *head* of r while the conjunction $D_1, \ldots, D_m$ is the *body* of r. The set of all basic conjunctions appearing in r is denoted by $H(r)$ while the set of all basic disjunctions appearing in r is denoted by $B(r)$. Moreover, the set of all positive basic disjunctions of r is denoted by $B^+(r)$ and the set of remaining basic disjunctions, called *non-positive*, is denoted by $B^-(r)$ (i.e. $B^-(r) = B(r) \setminus B^+(r)$).

A rule is *positive* if $B(r) = B^+(r)$. If all C_i are atoms and all D_j are literals respectively, the rule is called *standard*.

Example 2.1 (Normal Form Nested Rule)**.** *The following is a (normal form nested) rule:*

$$node(X), node(Y) \text{ :- } edge(X, Y).$$

The rule stands for "X is a node and Y is a node if there exists an edge between X and Y"*. The head of the rule is the positive basic conjunction* $node(X), node(Y)$ *while the body is the literal* $edge(X, Y)$*.*

An another example we consider the following rule:

$$a(X) \vee (b(X), c(X)) \text{ :- } d(X), (e(X) \vee \textbf{\textit{not}}\ f(Y)), (d(X) \vee s(Z) \vee f(X)).$$

where $d(X)$ *and* $(d(X) \vee s(Z) \vee f(X))$ *are positive basic disjunctions, while* $(e(X) \vee \textbf{\textit{not}}\ f(Y))$ *is non-positive, since it contains the negative literal* $\textbf{\textit{not}}\ f(Y)$*.*

Definition 2.4 (*NFN* Prgram)**.** *A Normal Form Nested (NFN program) program* P *is a finite set of rules.*

P is a *positive program* if all rules of P are positive. P is a *standard program* if all its rules are standard.

2.2 Semantics

We use the standard *DLP* approach of defining the semantics on its (Herbrand) instantiation by means of a reduct.

The instantiation on an *NFN* program is defined as in Section 1.2.1

Example 2.2. *Let* $U = \{1, 2\}$ *be a set of constants and* r *the rule:*

$$(a(X), b(Y)) \vee c(X) \text{ :- } (p(X, Y) \vee q(Y)), q(X).$$

The instantiation $Ground(r, U)$ *is the set containing the following rules:*

$$\begin{array}{l}(a(1), b(1)) \vee c(1) \text{ :- } (p(1,1) \vee q(1)), q(1).\\ (a(1), b(2)) \vee c(1) \text{ :- } (p(1,2) \vee q(2)), q(1).\\ (a(2), b(2)) \vee c(2) \text{ :- } (p(2,2) \vee q(2)), q(2).\\ (a(2), b(1)) \vee c(2) \text{ :- } (p(2,1) \vee q(1)), q(2).\end{array}$$

2.2.1 Interpretation and Models

As in Section 1.2.2, an *interpretation* I, for an *NFN* program P, is a set of ground atoms $I \subseteq B_P$, where B_P is the Herbrand base of P (see Definition 1.9).

Next we extend the Definition 1.15 from *DLP* to *NFN* with the following.

Definition 2.5. *Let P a program and I an interpretation for P, then the items* 1. *and* 2. *from Definition 1.15 hold. Moreover:*

N1. A ground basic disjunction L is true w.r.t. I if at least one literal of L is true w.r.t. I; otherwise L is false w.r.t. I.

N2. A ground basic conjunction A is true w.r.t. I if all atoms of A are true w.r.t. I; otherwise A is false w.r.t. I.

Let r be a ground NFN rule such that $r \in Ground(P)$:

N3. The head H of r is true w.r.t. I if at least one basic conjunction of H is true w.r.t. I; otherwise H is false w.r.t. I.

N4. The body B of r is true w.r.t. I if all basic disjunctions of B are true w.r.t. I; otherwise B is false w.r.t. I.

N5. r is satisfied w.r.t. I if the head is true w.r.t. I or the body is false w.r.t. I.

P is satisfied w.r.t. I if all rules of $Ground(P)$ are satisfied.

A *model* for P is an interpretation M for P such that $M \models P$. A model M for P is *minimal* if no model N for P exists such that $N \subsetneq M$.

2.2.2 Answer sets

Next we define a reduct for ground *NFN* programs w.r.t. an interpretation. It can be viewed as a generalization of the reduct defined in Definition 1.18 and a simplification of the one in [18].

Definition 2.6 (Reduct)**.** *Let P be a ground NFN program and I an interpretation. The reduct of P w.r.t. I, denoted by P^I, is defined as follows:*

(1) all rules with false body w.r.t. I are deleted;

(2) false body literals w.r.t. I from the remaining rules are deleted.

Given a rule $r \in P$, let r^I be the corresponding rule in P^I (which does not necessarily exist). Obviously r^I exists unless it was deleted in step (1).

Observation 1. *r^I exists iff $I \models B(r)$.*

Example 2.3. *Consider the following NFN program P:*

$a.$
$b.$
$f \vee (d, e)$:- $(a \vee \textbf{\textit{not}}\ c).$
p :- $(\textbf{\textit{not}}\ a \vee \textbf{\textit{not}}\ b).$
g :- $(b \vee \textbf{\textit{not}}\ a).$

and interpretation $I = \{a, b, f, g\}$, then P^I is the following program:

$a.$
$b.$
$f \vee (d, e)$:- $(a \vee \textbf{\textit{not}}\ c).$
g :- $b.$

Definition 2.7 (Answer set for *NFN* program)**.** *Given an NFN program P, an interpretation I is an answer set for P iff I is a minimal model for $Ground(P)^I$.*

The set of answer sets for P is denoted by $AS(P)$.

Example 2.4. *In Example 2.3, I is an answer set for the program P. Indeed, I is a model for P^I and it is easy to check that no subset $J \subsetneq I$ exists such that J satisfies all rules of P^I. Therefore*

$$AS(P) = \{a, b, d, e, g\}.$$

Looking at Definition 2.6, similar to the reduct for *DLP* programs defined in Definition 1.18, all rules with false body are deleted. Furthermore, from rule bodies of the remaining rules all false body literals are deleted. Without the latter deletion, in some cases it is possible to obtain unintuitive answer sets as shown in the following example.

Example 2.5. *Let us consider program*

$$P = \{c \text{ :- } (c \vee \textbf{\textit{not}}\ c).\}$$

and interpretation $I = \{c\}$.

If we just deleted all rules with false body w.r.t. I*, the reduct would be again* P *and* I *would be an answer set for* P*. However, using the reduct of Definition 2.6, we obtain*

$$\{c \text{ :- } c.\}$$

of which I *is not a minimal model as* $\emptyset$ *is also a model. Indeed,* I *is unintuitive as* c *is only justified by its own truth, and it is also not an answer set according to [18] (cf. Definition 6.1).*

2.3 Language Properties

Since the *NFN* programs generalize the class of *DLP* programs, in general an *NFN* program is not domain independent. Consequently, in this section we define a syntactic criterion for identifying *NFN* programs which are guaranteed to be domain independent.

2.3.1 Domain Independence

First of all we show some examples of domain dependent *NFN* programs.

Example 2.6. *Consider the program* P_u

$c(1).$
$d(1).$
$a(X) \vee b(Y) \text{ :- } (c(X) \vee d(Y)).$

where $U_{P_u} = \{1\}$*. Then* $Ground(P_u, U_{P_u})$ *is the following:*

$c(1).$
$d(1).$
$a(1) \vee b(1) \text{ :- } (c(1) \vee d(1)).$

We can verify

$$AS(Ground(P_u, U_{P_u})) = \{\{a(1), c(1), d(1)\}, \{b(1), c(1), d(1)\}\}.$$

Now consider $U = U_{P_u} \cup \{2\}$. *Then* $Ground(P_u, U)$ *is the following program:*

$a(1) \vee b(2)$:- $(c(1) \vee d(2))$.
$a(2) \vee b(1)$:- $(c(2) \vee d(1))$.
$a(2) \vee b(2)$:- $(c(2) \vee d(2))$.
$a(1) \vee b(1)$:- $(c(1) \vee d(1))$.
$c(1)$.
$d(1)$.

Then the set of answer sets of $Ground(P_u, U)$ *is*

$$\{\{a(1), a(2), c(1), d(1)\}, \quad \{a(1), b(1), c(1), d(1)\}, \quad \{b(1), b(2), c(1), d(1)\}\}$$

which means that we obtain different answer sets for the program P_u*, depending on the considered universe. So, for instance when a completely unrelated fact* $x(2)$ *is added to* P_u*, this will unexpectedly cause additional atoms containing predicates* a *and* b *to become true in its answer sets. The problem is that in the main rule of* P_u *the body can be satisfied without "binding" one of the two variables that occur in the head.*

Example 2.7. *Consider the program* P_{u2}*:*

$c(1)$.
a :- $(b(X) \vee \textbf{\textit{not}}\ c(X))$.

where $U_{P_{u2}} = \{1\}$. *The ground program* $Ground(P_{u2}, U_{P_{u2}})$ *is*

$c(1)$.
a :- $(b(1) \vee \textbf{\textit{not}}\ c(1))$.

so the program has one answer set

$$AS(Ground(P_{u2}, U_{P_{u2}})) = \{c(1)\}.$$

Now if $U_2 = U_{P_{u2}} \cup \{2\}$, $Ground(P_{u2}, U_2)$ *is*

$c(1)$.
a :- $(b(1) \vee \textbf{\textit{not}}\ c(1))$.
a :- $(b(2) \vee \textbf{\textit{not}}\ c(2))$.

which admits

$$AS(Ground(P_{u2}, U_2)) = \{a, c(1)\}.$$

So here adding a completely unrelated fact $x(2)$ *to the program would unexpectedly give a reason to assume* a*. In this case the variable in the negative literal does not necessarily have a restriction when the body is true.*

Now we give the formal definition of domain independence for an *NFN* program, stating that the semantics should be independent of the universe, as long as it is sufficiently large.

Definition 2.8. *Let P be an NFN program and U_P be the set of constants appearing in P. P is* domain independent *if for each $U \supseteq U_P$,*

$$AS(Ground(P, U)) = AS(Ground(P, U_P))$$

holds.

Examples 2.6 and 2.7 directly serve as a proof for the following fact.

Fact 2.3.1. *An NFN program P is in general not domain independent.*

Note that, since *NFN* programs contain the class of *DLP* programs, the previous result follows from Facts 1.3.1.

2.3.2 Safe Programs

In this section, we will define a class of *NFN* programs, which is characterized syntactically, such that all programs in this class are guaranteed to be domain independent.

Definition 2.9 (Safe variable)**.** *Let r be an NFN rule. A variable $X \in$* vars(r) *is* safe *if there exists a positive basic disjunction $D \in B(r)$, such that for each atom a in D, $X \in$* vars(a)*; we also say that D saves X or X is made safe by D.*

Example 2.8. *Consider the rule*

$$a \text{ :- } (b(X) \vee c(X, Z) \vee d(X)), e(Y), (s(Z) \vee t(X)).$$

The safe variables of the rule are X and Y. Indeed, the variable X is safe because it appears in all atoms of the positive basic disjunction $D_1 = (b(X) \vee c(X, Z) \vee d(X))$, while the variable Y occurs in the only atom of the positive basic disjunction $e(Y)$.

Definition 2.10 (Safe rules and programs)**.** *An NFN rule r is safe if each variable that appears in the head of r and each variable that appears in some negative body literal of r is safe. An NFN program P is safe if each rule is safe.*

Example 2.9. *The NFN rule*

$$h \text{ :- } (a(X) \vee b(X)), \textbf{\textit{not}}\ c(X).$$

is safe. In fact, variable X, which appears in the negative literal ***not*** *$c(X)$ is made safe by $(a(X) \vee b(X))$.*

The rule

$$(h_1(X), h_2(X)) \text{ :- } (a(X) \vee b(Z)), (c(X) \vee \textbf{\textit{not}}\ s(Z)).$$

is not safe, since the variable X occurs in the head of the rule but no positive basic disjunction in the body saves X. Moreover, variable Z, occurring in negative body literal ***not*** *$s(Z)$, is also unsafe.*

In the follows, we show that the class of safe programs is domain independent.

Lemma 2.3.1. *Let r be a safe NFN rule, I a set of ground atoms, $U \supseteq const(I)$ and $U' \supset U$, then*

$$I \models Ground(r, U) \Rightarrow I \models Ground(r, U').$$

Proof. Assume $I \models Ground(r, U)$ and $I \not\models Ground(r, U')$. Then, a substitution

$$\sigma : vars(r) \rightarrow U'$$

exists s.t. $I \not\models r\sigma$, so

$$I \models B(r\sigma) \quad and \quad I \not\models H(r\sigma).$$

Since $I \models B(r\sigma)$, then for each positive basic disjunction $D_p \in B^+(r)$ an atom $a \in D_p$ exists s.t. $a\sigma \in I$ so $const(a\sigma) \subseteq U$. Moreover, for each non-positive basic disjunction $D_n \in B^-(r)$ a literal $l \in D_n$ exists s.t. $I \models l\sigma$. Therefore, if l is positive, $l\sigma \in I$ and $const(l\sigma) \subseteq U$; if $l = \textbf{not}\ a$, $a\sigma \notin I$ and since r is safe, for all $X \in vars(a)$ a positive disjunction $D_s \in B^+(r)$ exists s.t. for all $\bar{a} \in D_s$, $X \in vars(\bar{a})$ and thus $const(l\sigma) \subseteq U$. Then, arbitrarily choose a substitution $\sigma' : vars(r) \mapsto U$ such that for all $X/c \in \sigma$ s.t. $c \in (U' \setminus U)$ there exists $s \in U$ s.t. $X/s \in \sigma'$, and for all $X/c \in \sigma$ s.t. $c \in U$, $X/c \in \sigma'$. Then it holds that $I \models B(r\sigma')$ and $I \models H(r\sigma')$ because $r\sigma' \in Ground(P, U)$. Furthermore, r is safe so for all $X \in vars(H(r))$ a positive disjunction $D_p \in B^+(r)$ exists s.t. for all $\bar{a} \in D_p$, $X \in vars(\bar{a})$ and thus $const(H(r\sigma)) \subseteq U$, hence $H(r\sigma') = H(r\sigma)$, and we obtain a contradiction. □

Lemma 2.3.2. *Let r be a safe NFN rule, I a set of ground atoms, and $U \supseteq const(I)$, then $Ground(r,U)^I = Ground(r,U')^I$ for all $U' \supset U$.*

Proof. Since $U' \supset U$, from Definition 1.13, $Ground(r,U) \subseteq Ground(r,U')$ and therefore, $Ground(r,U)^I \subseteq Ground(r,U')^I$.

Let $\bar{r} \in Ground(r,U')^I$ and assume $\bar{r} \notin Ground(r,U)^I$ then there exists a literal $l \in B(\bar{r})$ s.t. $I \models l$ (from Definition 2.6) and $const(l) \cap (U' \setminus U) \neq \emptyset$. If l is positive, $l \in I$ and thus $const(l) \subseteq U$. If $l =$ **not** a then $a \notin I$ but since r is safe, for all $X \in vars(a)$ a positive disjunction $D_p \in B^+(r)$ exists s.t. for all $\bar{a} \in D_p$, $X \in vars(\bar{a})$ and thus $const(l) \subseteq U$. So we have a contradiction with the hypothesis $const(l) \cap (U' \setminus U) \neq \emptyset$. □

Theorem 2.3.3. *If P is safe then P is domain independent.*

Proof. $\forall U \supseteq U_P : AS(Ground(P,U_P)) \subseteq AS(Ground(P,U))$:
Let P be a safe program and assume that $I \in AS(Ground(P,U_P))$ and $\exists U \supset U_P$ s.t. $I \notin AS(Ground(P,U))$. (i) If I is not a model of $Ground(P,U) \Rightarrow \exists r \in P$ and a substitution $\sigma : vars(r) \mapsto U$ s.t. $I \not\models r\sigma$. Since $I \models Ground(P,U_P) = \bigcup_{r' \in P} Ground(r',U_P)$, for each $r' \in P$ $I \models Ground(r',U_P)$ and from Lemma 2.3.1, $I \models Ground(r',U)$. Then $I \models r\sigma$ for each $r\sigma \in Ground(P,U)$ and we obtain a contradiction. (ii) If I is a model for $Ground(P,U)$ but I is not a minimal model for $Ground(P,U)^I$, then $\exists J \subset I$ s.t. J is a model for $Ground(P,U)^I$. Since $Ground(P,U_P) \subset Ground(P,U)$, then $(Ground(P,U_P)^I) \subseteq (Ground(P,U)^I)$ and J is a model for $Ground(P,U_P)^I$ contradicting $I \in AS(Ground(P,U_P))$.

$\forall U \supseteq U_P : AS(Ground(P,U)) \subseteq AS(Ground(P,U_P))$:
Let P be a safe program and assume that $\exists U \supset U_P$ s.t. $I \in AS(Ground(P,U))$ and $I \notin AS(Ground(P,U_P))$. (i) Since $Ground(P,U_P) \subset Ground(P,U)$ then I is a model for $Ground(P,U_P)$. (ii) If I is a model for $Ground(P,U_P)$ but I is not a minimal model for $Ground(P,U_P)^I$ then there exists $J \subset I$ s. t. $J \in AS(Ground(P,U_P)^I)$. By Lemma 2.3.2 it holds that $Ground(P,U_P)^I = Ground(P,U)^I$, and therefore $J \models Ground(P,U)^I$ and this is a contradiction to $I \in AS(Ground(P,U))$. □

Chapter 3

Knowledge Representation by *NFN* Programs

Disjunctive Logic Programming under answer set semantics has been proved to be a very effective formalism for *Knowledge Representation and Reasoning (KRR)*. It can be used to encode problems in a highly declarative fashion, following the "Guess&Check"(**G&C**) methodology presented in [4]. Applying this technology using *NFN* encodings we can obtain a more compact and intuitive representation w.r.t. *DLP* encodings.

The chapter is organized as follows:

- ★ In Section 3.1 we describe the **G&C** technique.
- ★ In Section 3.2 we illustrate how to apply it on a number of examples by using *NFN* programs.

3.1 The Guess and Check Programming Methodology

Many problems, also problems of comparatively high computational complexity (Σ_2^P-complete and Π_2^P-complete problems), can be solved in a natural manner by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so in a uniform way over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a **G&C** program $\mathcal{P}$ for $\mathbf{P}$ consists of the following two main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance I.

Without imposing restrictions on what kind of rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., checking is completely integrated into the guessing part such that solution candidates are always solutions. Also, in general, the search space may be limited by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

In general, both $\mathcal{G}$ and $\mathcal{C}$ may be arbitrary collections of rules, and it depends on the complexity of the problem at hand which kinds of rules are needed to implement these parts (in particular, the checking part).

For problems with complexity in NP, often a natural **G&C** program can be designed with the two parts clearly separated into the following simple layered structure:

- The guessing part $\mathcal{G}$ consists of disjunctive rules that "guess" a solution candidate S.

- The checking part $\mathcal{C}$ consists of integrity constraints that "check" the admissibility of S.

Each layer may have further auxiliary predicates, for local computations. The disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

It is worth remarking that the **G&C** programming methodology has also positive implications from the Software Engineering point of view. Indeed, the modular program structure in **G&C** allows for developing programs incrementally, which is helpful to simplify testing and debugging. One can start by writing the guessing part $\mathcal{G}$ and testing that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one adds the checking part and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ encode the admissible solutions.

3.2 Applications of the Guess and Check Technique

In this section, we illustrate the declarative programming methodology described in Section 3.1 by showing its application on a number of concrete examples.

3.2.1 Connected Monochromatic Triangle

Let us consider a variation of the NP-complete problem *Monocromatic Triangle* discussed by Garey and Johnson in [10].

Definition 3.1. *Let $G = (V, E)$ be an undirected graph where V is a set of vertices and E is a set of edges. G is a connected monochromatic triangle if there exists a partition of E in two disjoint sets E_1 and E_2 such that for $V_i = \{a, b | (a, b) \in E_i\}$, $i = \{1, 2\}$, the following conditions hold:*

1. *E_1 and E_2 are non-empty;*
2. *both graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are connected;*
3. *neither of the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ contain a triangle.*

We can ignore isolated vertices because we are interested in partitions of edges, therefore we suppose that the input graph G is specified only by atoms $edge(X, Y)$, denoting that there is an edge from vertex X to vertex Y. The following **G&C** program P_{mt} solves the problem.

GUESS: $inPart(X, Y, 1) \vee inPart(X, Y, 2)$:- $edge(X, Y)$.
CHECK:

1. $\begin{cases} hasElements(P) \text{ :- } inPart(\_, \_, P). \\ \text{ :- } \textbf{not } hasElements(1) \vee \textbf{not } hasElements(2). \end{cases}$

2. $\begin{cases} vtxP(X, P), vtxP(Y, P) \text{ :- } inPart(X, Y, P). \\ reaches(X, Y, P) \text{ :- } vtxP(X, P), vtxP(Y, P), X <> Y, \\ \qquad (inPart(X, Z, P) \vee inPart(Z, X, P)), X <> Z, \\ \qquad (Y = Z \vee reaches(Z, Y, P) \vee reaches(Y, Z, P)). \\ \text{ :- } vtxP(X, P), vtxP(Y, P), X <> Y, \textbf{not } reaches(X, Y, P). \end{cases}$

3. $\begin{cases} \text{ :- } (inPart(X, Y, P) \vee inPart(Y, X, P)), X <> Y, \\ \quad (inPart(Z, Y, P) \vee inPart(Y, Z, P)), Y <> Z, \\ \quad (inPart(X, Z, P) \vee inPart(Z, X, P)), X <> Z. \end{cases}$

In the **GUESS** part, the *NFN* rule guesses the bipartition of E, represented by predicate $inPart$. Note that $inPart$ is completely determined by this rule and no further guessing is needed. Moreover, the rule assures that all input edges are considered.

In the **CHECK** part, the set of rules in each item represents the corresponding condition described in Definition 3.1. In detail in 1. it is checked that each partition is non-empty. In 2. it is checked that each induced subgraph is connected. In detail, predicate $vtxP$ represents the set of vertexes in a partition P. The second rule defines the reachability of two vertexes and the constraint enforces that each pair of vertices in a partition are reachable. In the last item the constraint ensures that each induced graph does not contain triangle.

It is easy to see that each partition of edges represented by answer sets of P_{mt} is a bipartition of G satisfying all conditions. If P_{mt} does not admit any answer set the input graph is not a connected monochromatic triangle. Therefore, given a set of facts F for $edge$, the program $P_{mt} \cup F$ has an answer set if and only if the input graph is a connected monochromatic triangle.

3.2.2 The game of Marriage (Bipartite Matching)

The *game of marriage* is related to the *Hall's Theorem* also known as the marriage theorem [14].

Definition 3.2. *In the game of marriage there are n single women and m single men who desire to get married. Therefore each person indicates who among the opposite sex would be acceptable as a potential spouse. X and Y get married if X indicates only Y as a potential spouse and Y indicates only X as a potential spouse.*

The input is represented by atoms $p(Name, Gender)$ ($Gender \in \{m, w\}$, m for man and w for woman) representing persons, and atoms $prefer(X, Y)$ denoting that X indicates Y as a potential spouse.

The following **G&C** program, P_{mg} solves the problem.

GUESS: $wed(X, Y), wed(Y, X) \vee noWed(X, Y)$:- $p(X, w), p(Y, m)$.

CHECK
- :- $prefer(X, Y), prefer(Y, X),$ **not** $wed(X, Y)$.
- :- $wed(X, Y), ($**not** $prefer(X, Y) \vee$ **not** $prefer(Y, X))$.
- :- $wed(X, Y), (prefer(X, Z) \vee prefer(Y, Z)), X <> Z, Y <> Z$.

The rule in the **GUESS** part guesses all possible pairs of spouses. The first constraint in the **CHECK** part guarantees that if X indicated Y as a potential spouse and Y indicated X as a potential spouse than X and Y wed. The second constraint guarantees that if X did not prefer Y or vice versa, X cannot wed Y. Finally, the third contraint assures that if X or Y expressed at least two preference than X cannot wed Y.

Let F be a set of facts for p and $prefer$, then $F \cup P_{mg}$ has an answer set if and only if there exists at least a pair of persons X, Y such that X prefers only Y as a potential spouse and vice versa.

We can represent this situation also by a *bipartite graph*. These graphs are both useful and common so we give a full representation in the following section.

3.2.3 Bipartite Graph

Let us consider another problem in the graph theory, namely *Bipartite Graph* [19].

Definition 3.3 (Bipartite Graph)**.** *Given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, G is a Bipartite Graph if V can be divided into two disjoint sets V_1 and V_2, such that the following conditions hold:*

1. *V_1 and V_2 are not empty;*
2. *for each $v_i \in V_i$ there exists at least one edge connecting v_i and v_j, $v_j \in V_j$, $i, j \in \{1, 2\}$, $i \neq j$;*
3. *for each $\{v_i, v_j\} \in V_i$, $i \in \{1, 2\}$ no edge connecting v_i and v_j exists.*

The input graph G is specified by atoms $vtx(X)$ denoting that X is a vertex of G and by atoms $edge(X, Y)$, denoting that there is an edge from vertex X to vertex Y. The **G&C** program P_{bp} representing the problem is the following:

GUESS: $pVtx(X, 1) \vee pVtx(X, 2)$:- $vtx(X)$.

CHECK:

1. $\begin{cases} hasElements(P) \text{ :- } pVtx(X, P). \\ \text{ :- } \textbf{not } hasElements(1) \vee \textbf{not } hasElements(2). \end{cases}$

2. $\begin{cases} connected(X, P) \text{ :- } pVtx(X, P), pVtx(Y, P2), P <> P2, \\ \qquad\qquad\qquad\qquad (edge(X, Y) \vee edge(Y, X)). \\ \text{ :- } pVtx(X, P), \textbf{not } connected(X, P). \end{cases}$

3. $\{$:- $pVtx(X, P), pVtx(Y, P), X <> Y, edge(X, Y)$.

The rule in the **GUESS** part guesses subsets of vertices. In the **CHECK** part, the set of rules in each item represents the corresponding condition described in Definition 3.3. In particular, in item 1. the constraint assures that each subset is nonempty. In the second item the constraint guarantees that there exists at least an edge connecting a node in a partition with a node in the another partition. The last constraint assures that no edge connects nodes that are in the same partition.

Chapter 4

Translations From *NFN* to *DLP* Programs

In this chapter we describe translations from Normal Form Nested programs into disjunctive logic programs. This allows for evaluating ground nested logic programs using existing disjunctive logic programming systems as a back-end.

The chapter is organized as follows:

- ★ In Section 4.1 we describe a transformation which may produce a program which is exponentially larger than the input program. The answer sets of the *DLP* program are exactly the answer sets of the original program and it maintains also safety.
- ★ In Section 4.2 we discuss a polynomial translation, inspired by structure-preserving normal form translations, that works for ground programs only. This method introduces new symbols implying that the result is not equivalent in the classical sense to the original program, but there is a one-to-one correspondence between the answer sets.
- ★ In Section 4.3 we present an algorithm that efficiently translates *NFN* programs with variables to *DLP* programs. We show that this algorithm also maintains safety, there is a one-to-one correspondence between answer sets, and the size of the generated program is bounded by a small polynomial of the original *NFN* program size.

4.1 Naïve Transformation

In this section we describe a straightforward method to translate normal form nested programs into disjunctive logic programs. This method is based on distributivity transformations and does not introduce new atoms. It preserves the answer sets of the original program. Moreover, if the *NFN* program is safe also the corresponding *DLP* program is safe. Unfortunately, as common for this type of transformation and as shall be seen by an example, this transformation may exponentially enlarge the program.

Definition 4.1. *Let $H = C_1 \vee \cdots \vee C_m$, $m \geq 0$ be a formula in disjunctive normal form where $C_i = a_{i1}, \ldots, a_{in_i}$, $n_i \geq 0$. The corresponding list of basic disjunctions H° of H is obtained first replacing H by $a_{11} \vee C_2 \vee \cdots \vee C_m; \ldots; a_{1n_1} \vee C_2 \vee \cdots \vee C_m$, and then iterating on each C_i, $i \geq 2$. As a result:*

$$H^\circ = a_{11} \vee \ldots \vee a_{m1};\ \ldots;\ a_{1n_1} \vee \ldots \vee a_{mn_m}$$

where $|H^\circ|$ has $n_1 \times n_2 \times \ldots \times n_m$ disjunctions.

Example 4.1. *Let H be the formula $a(X), b(X) \vee c(Y), d(Y)$, as first step we obtain*

$$a(X) \vee (c(Y), d(Y));\ b(X) \vee (c(Y), d(Y))$$

and as final step:

$$H^\circ = a(X) \vee c(Y);\ a(X) \vee d(Y);\ b(X) \vee c(Y); b(X) \vee d(Y)$$

and $|H^\circ|$ has 2×2 disjunctions.

Definition 4.2. *Let $B = D_1, \ldots, D_m$, $m \geq 0$ be a formula in conjunctive normal form where $D_i = l_{i1} \vee \ldots \vee l_{in_i}$, $n_i \geq 0$. The corresponding list of basic conjunction B° of B is obtained first replacing B by $l_{11}, D_2, \ldots, D_m; \ldots; l_{1n_1}, D_2, \ldots, D_m$, and then iterating on each D_i, $i \geq 2$. As a result:*

$$B^\circ = l_{11}, \ldots, l_{m1};\ \ldots;\ l_{1n_1}, \ldots, a_{mn_m}$$

where $|B^\circ|$ has $n_1 \times n_2 \times \ldots \times n_m$ conjunctions.

Example 4.2. *Let B be the formula $e(X,Y) \vee f(X,Y), g(X) \vee s(Z)$, as first step we obtain*

$$e(X,Y), (g(X) \vee s(Z));\ f(X,Y), (g(X) \vee s(Z)).$$

Therefore, as final step:

$$B^\circ = e(X,Y), g(X); e(X,Y), s(Z);\ f(X,Y), g(X);\ f(X,Y), s(Z)$$

and B° has 2×2 conjunctions.

Definition 4.3. *Let r be an NFN rule, $H^\circ(r) = H_1; \ldots; H_n$ be the corresponding list of basic disjunctions of $H(r)$ built as defined in Definition 4.1, and $B^\circ(r) = B_1; \ldots; B_m$ the corresponding list of basic conjunctions of $B(r)$ built as defined in Definition 4.2. The DLP program for r is defined as*

$$P_r = \{H_i \text{ :- } B_j. |\ \forall i = 1, \ldots, n \text{ and } \forall j = 1, \ldots, m\}.$$

For an NFN program P the naïve transformation P_{exp} is

$$P_{exp} = \bigcup_{r \in P} P_r. \tag{4.1}$$

Example 4.3. *Let P be the following NFN program:*

$$r_1 :\ a(X), b(X) \vee c(Y), d(Y) \text{ :- } e(X,Y) \vee f(X,Y).$$
$$r_2 :\ e(1,1).$$

Therefore

$$H^\circ = a(X) \vee c(Y);\ a(X) \vee d(Y);\ b(X) \vee c(Y); b(X) \vee d(Y);$$

as shown in Example 4.1. Moreover $B^\circ(r) = e(X,Y); f(X,Y)$. Consequently, the corresponding DLP program P_{exp} of P is:

$$\begin{array}{ll} a(X) \vee c(Y) \text{ :- } e(X,Y). & a(X) \vee c(Y) \text{ :- } f(X,Y). \\ a(X) \vee d(Y) \text{ :- } e(X,Y). & a(X) \vee d(Y) \text{ :- } f(X,Y). \\ b(X) \vee c(Y) \text{ :- } e(X,Y). & b(X) \vee c(Y) \text{ :- } f(X,Y). \\ b(X) \vee d(Y) \text{ :- } e(X,Y). & b(X) \vee d(Y) \text{ :- } f(X,Y). \\ e(1,1). & \end{array}$$

Note that

$$AS(P) = \{\{e(1,1), a(1), b(1)\}, \{e(1,1), c(1), d(1)\}\} = AS(P_{exp}).$$

4.1.1 Properties of the naïve Transformation

Next we report some properties of the naïve transformation. The first one is rather important and states that the naïve transformation maintains safety.

Proposition 4.1.1. *Given a safe NFN program P, then also P_{exp} is safe.*

Proof. In P, a rule r must be safe, that is each variable occurring in $H(r)$ or in a negative body literal must occur in each atom of a positive basic disjunction D of $B(r)$. Now note that each element of $B(r)^\circ$ contains one literal of each basic disjunction of D. Therefore, each variable that occurs in $H(r)$ or in a negative body literal of r also occurs in a positive literal (stemming from D) in each rule of P_r. Since all head atoms or negative body literals of rules in P_r have also been head atoms or negative body literals in r, P_r is safe. Since these considerations hold for all rules in P, also P_{exp} is safe. □

We next want to show that the transformation also maintains answer sets. In order to do this, we first show a preliminary result.

Lemma 4.1.2. *Given an NFN rule r, a set of constants U, and an interpretation I for $Ground(r, U)$, $I \models Ground(r, U)$ iff $I \models Ground(P_r, U)$.*

Proof. We first observe that for any substitution σ, $P_r\sigma = P_{r\sigma}$. In the following, let $s = r\sigma$ for an arbitrary substitution $\sigma : vars(r) \to U$. We show $I \models s \Leftrightarrow I \models P_s$, from which $I \models Ground(r, U) \Leftrightarrow I \models Ground(P_r, U)$ follows.

($\Rightarrow$) Assume $I \models s$, and first assume $I \not\models B(s)$. Then there exists a basic disjunction $D \in B(s)$ such that for each literal $\ell \in D$, $I \not\models \ell$. We observe that by construction each rule $t \in P_s$ contains a literal of D in its body and hence $I \not\models B(t)$ and $I \models t$, thus $I \models P_s$. Now assume $I \models H(s)$. This means that there exists a basic conjunction $C \in H(s)$ such that for each atom $a \in C$, $I \models a$. Again, by construction each rule $t \in P_s$ contains an atom of C in its head and hence $I \models H(t)$, which implies $I \models t$ and thus $I \models P_s$.

($\Leftarrow$) Assume $I \models P_s$, and first consider the case in which $I \not\models B(t)$ for all $t \in P_s$. So there exists a literal $\ell_t \in B(t)$ such that $I \not\models \ell_t$ for each $t \in P_s$. Due to the properties of CNF-to-DNF conversion, there is at least one basic disjunction in $B(s)$, in which all literals are false w.r.t. I, whence $I \not\models B(s)$

and thus $I \models s$. Now consider that $I \models B(t)$ for some $t \in P_s$, then $I \models H(t)$ as well. Note that P_s contains a rule having body $B(t)$ for each H_i of the sequence H° of Definition 4.3, and so $I \models H_i$ for all $1 \leq i \leq n$. This implies that there must be a basic conjunction C in $H(s)$ such that for each a in C $I \models a$, whence $I \models C$, $I \models H(s)$ and $I \models s$ follows. □

Corollary 4.1.3. *Given an NFN program P, a set of constants U, and an interpretation I, $I \models Ground(P,U)$ iff $I \models Ground(P_{exp},U)$.*

Lemma 4.1.4. *Given a ground NFN program P and an interpretation I for $Ground(P,U)$, $(P^I)_{exp} = (P_{exp})^I$.*

Proof. An $r \in P$ for which $I \not\models B(r)$ is not in P^I. For these r there is a basic disjunction with all literals being false w.r.t. I. Since at least one of these literals is contained in each $t \in P_r$ also $I \not\models B(t)$ for all $t \in P_r$. So if r^I does not exist, $(P_r)^I = \emptyset$. If $I \models B(r)$, then each rule $t \in P_r$ that contains a false literal w.r.t. I in the body will not be in $(P_r)^I$. But since these literals are not in r^I, also P_{r^I} will not contain t. All other rules in P_r will be in both $(P_r)^I$ and P_{r^I}. In total, this implies $(P^I)_{exp} = (P_{exp})^I$. □

Proposition 4.1.5. *Given a safe NFN program P, $AS(P) = AS(P_{exp})$.*

Proof. $I \in AS(P)$ iff $I \models Ground(P)^I$ and $J \not\models Ground(P)^I$ for each $J \subsetneq I$. By Corollary 4.1.3 this is equivalent to $I \models (Ground(P)^I)_{exp}$ and $J \not\models (Ground(P)^I)_{exp}$ for each $J \subsetneq I$. By Lemma 4.1.4 $(Ground(P)^I)_{exp} = (Ground(P)_{exp})^I$ and, since $Ground(P)_{exp} = Ground(P_{exp})$, the statement is equivalent to $I \models Ground(P_{exp})^I$ and $J \not\models Ground(P_{exp})^I$ for each $J \subsetneq I$ which is the definition for $I \in AS(P_{exp})$. □

Proposition 4.1.6. *Let P be a safe NFN program and P_{DLP} the corresponding standard program built by means of the naïve transformation, then P_{DLP} has exponential size in $|P|$.*

Proof. Let m the total number of rules in P, let $n = \max\{n_d, n_c\}$ where n_d is the maximum number of basic disjunctions and n_c the maximum number of basic conjunctions in a rule. Moreover, let $j = \max\{j_d, j_c\}$ where j_d is the maximum number of literals in a basic disjunction and j_c the maximum number of atoms in a basic conjunction. Then $|P_{DLP}| \leq m * j^{2n}$ and it is easy to find programs for which the equality holds. □

4.2 Efficient Translation from Ground *NFN* to Ground *DLP* programs

While the naïve transformation has desirable properties, such as maintaining safety and answer sets, its exponential space behavior makes it impractical. Similar observations have been made for normal form transformations in classical logic and transformations have been suggested, which avoid the exponential increase and preserve the formula structure by introducing additional symbols.

In this section we describe the polynomial translation from ground normal form nested programs into ground disjunctive logic programs proposed by Pearce et al. in [20]. This translation is derived from *structure-preserving normal form translations* [21] and it introduces new labels, abbreviating subformula occurrences. Moreover we propose an optimized definition of this translation for *NFN* programs where a more succinct standard program is built.

In the following, for any set of predicate symbols Π (see Section 1.1), we use a new and disjoint alphabet $\Pi_{\mathbf{L}} = \{\mathbf{L}_\phi | \phi \in \Pi\}$ of labels.

Definition 4.4 ([20])**.** *Let P be a ground NFN program, then the corresponding DLP program $\sigma(P)$ is defined as follows:*

$$\sigma(P) = \{\mathrm{L}_{H(r)} \text{ :- } \mathrm{L}_{B(r)}. \,|r \in P\} \cup \gamma(P),$$

where $\gamma(P)$ is constructed as follows:

1. *for each positive literal occurring in P, γ contains the three rules*

$$l \text{ :- } \mathbf{L}_l. \qquad \mathbf{L}_l \text{ :- } l.$$

2. *for each negative literal **not** l occurring in P, γ contains the two rules:*

$$l_n \text{ :- } \mathbf{L}_l. \qquad \mathbf{L}_l \text{ :- } \textbf{\textit{not}}\ l. \qquad \text{:- } l_n, l.$$

3. *for each expression $\phi = (\phi_1, \phi_2)$ occurring in P, γ contains the three rules*

$$\mathbf{L}_\phi \text{ :- } \mathbf{L}_{\phi_1}, \mathbf{L}_{\phi_2}. \qquad \mathbf{L}_{\phi_1} \text{ :- } \mathbf{L}_\phi. \qquad \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_\phi.$$

4. *for each expression $\phi = (\phi_1 \vee \phi_2)$ occurring in P, γ contains the three rules*

$$\mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_\phi. \qquad \mathbf{L}_\phi \text{ :- } \mathbf{L}_{\phi_1}. \qquad \mathbf{L}_\phi \text{ :- } \mathbf{L}_{\phi_2}.$$

Example 4.4. *Let us consider the following NFN program P*

$$r_1: \quad (a,b) \vee (c,d) \text{ :- } f \vee g. \qquad r_2 : f.$$

with predicate symbols of the alphabet $\mathcal{A}_1 = \{a, b, c, \ldots, z\}$. Then $\sigma(P)$ is

$$\begin{array}{llllll} a \text{ :- } \mathbf{L}_a. & \mathbf{L}_a \text{ :- } a. & b \text{ :- } \mathbf{L}_b. & \mathbf{L}_b \text{ :- } b. & c \text{ :- } \mathbf{L}_c. & \mathbf{L}_c \text{ :- } c. \\ d \text{ :- } \mathbf{L}_d. & \mathbf{L}_d \text{ :- } d. & f \text{ :- } \mathbf{L}_f. & \mathbf{L}_f \text{ :- } f. & g \text{ :- } \mathbf{L}_g. & \mathbf{L}_g \text{ :- } g. \end{array}$$

$$\begin{array}{lll} \mathbf{L}_{a,b} \text{ :- } \mathbf{L}_a, \mathbf{L}_b. & \mathbf{L}_a \text{ :- } \mathbf{L}_{a,b}. & \mathbf{L}_b \text{ :- } \mathbf{L}_{a,b}. \\ \mathbf{L}_{c,d} \text{ :- } \mathbf{L}_c, \mathbf{L}_d. & \mathbf{L}_c \text{ :- } \mathbf{L}_{c,d}. & \mathbf{L}_d \text{ :- } \mathbf{L}_{c,d}. \end{array}$$

$$\begin{array}{lll} \mathbf{L}_f \vee \mathbf{L}_g \text{ :- } \mathbf{L}_{B(r_1)}. & \mathbf{L}_{B(r_1)} \text{ :- } \mathbf{L}_f. & \mathbf{L}_{B(r_1)} \text{ :- } \mathbf{L}_g. \\ \mathbf{L}_{a,b} \vee \mathbf{L}_{c,d} \text{ :- } \mathbf{L}_{H(r_1)}. & \mathbf{L}_{H(r_1)} \text{ :- } \mathbf{L}_{a,b}. & \mathbf{L}_{H(r_1)} \text{ :- } \mathbf{L}_{c,d}. \end{array}$$

$$\mathbf{L}_{H(r_1)} \text{ :- } \mathbf{L}_{B(r_1)}. \qquad \mathbf{L}_f.$$

where

$$\{a, b, c\} \in \Pi$$

and

$$\{\mathbf{L}_a, \mathbf{L}_b, \mathbf{L}_c, \mathbf{L}_d, \mathbf{L}_f, \mathbf{L}_g, \mathbf{L}_{a,b}, \mathbf{L}_{c,d}, \mathbf{L}_{H(r_1)}, \mathbf{L}_{B(r_1)}\} \in \Pi_{\mathbf{L}}.$$

It is quite obvious that the program $\sigma(P)$ is constructible in polynomial time.

Theorem 4.2.1 ([20])**.** *Let P be a ground NFN program with predicate symbols in Π and $\sigma(P)$ the corresponding DLP program built according to the Definition 4.4 with predicate symbols in $\Pi_{\mathbf{L}} \cup \Pi$, then*

1. *the time required to compute $\sigma(P)$ is polynomial in the size of P;*
2. *$AS(P) = \{I \cap \Pi \mid I \in AS(\sigma(P))\}$;*
3. *$\sigma(P \cup P_1) = \sigma(P) \cup \sigma(P_1)$, where P_1 is a ground NFN program.*

Example 4.5. *Let us consider the NFN program P in the Example 4.4, the answer sets of $\sigma(P)$ are:*

$$AS_1 = \{\mathbf{L}_f, f, \mathbf{L}_{B(r_1)}, \mathbf{L}_{H(r_1)}, \mathbf{L}_{a,b}, \mathbf{L}_a, \mathbf{L}_b, a, b\};$$

$$AS_2 = \{\mathbf{L}_f, f, \mathbf{L}_{B(r_1)}, \mathbf{L}_{H(r_1)}, \mathbf{L}_{c,d}, \mathbf{L}_c, \mathbf{L}_d, c, d\}.$$

Therefore $AS_1 \cap \Pi = \{f, a, b\}$ and $AS_2 \cap \Pi = \{f, c, d\}$ and it easy to see that $\{f, a, b\}$ and $\{f, c, d\}$ are the answer sets of P.

Besides the particular type of translation defined in Definition 4.4, there are also other, slightly improved structure-preserving normal form translations in which fewer rules are introduced, depending on the polarity of the corresponding subformula occurrences. (see Appendix A).

4.2.1 Optimized Efficient Translation for Ground *NFN* Programs

In what follows, we propose a variant of the translation in Definition 4.4 building a smaller *DLP* program.

Definition 4.5. *Let P be a ground NFN program and*

$$r = C_1 \vee \ldots \vee\ C_n \text{ :- } D_1, \ldots, D_m., \quad n, m \geq 0$$

be a rule of P, then the corresponding DLP program of r, $\sigma_{opt}(r)$, is defined as follows:

$$\sigma_{opt}(r) = \{\overline{C}_1 \vee \ldots \vee \overline{C}_n \text{ :- } \overline{D}_1, \ldots, \overline{D}_m. \,|r \in P\} \cup \gamma_{opt}(r),$$

where $\overline{C}_i$ is C_i if it is a literal, otherwise it is $\mathbf{L}_{C_i}$, $i = 1, \ldots, n$. Similarly, $\overline{D}_j$ is D_j if it is a literal, otherwise it is $\mathbf{L}_{D_j}$, $j = 1, \ldots, m$. Moreover, the set of rules $\gamma_{opt}(r)$ is constructed as follows:

- *for each basic conjunction $C = (\phi_1, \ldots, \phi_k)$, $k > 1$, occurring in r, γ_{opt} adds the $k + 1$ rules*

$$\mathbf{L}_C \text{ :- } \phi_1, \ldots, \phi_k. \qquad \phi_1 \text{ :- } \mathbf{L}_C. \qquad \ldots \qquad \phi_k \text{ :- } \mathbf{L}_C.$$

- *for each basic disjunction $D = (\phi_1 \vee \ldots \vee \phi_k)$ occurring in the body of some rules of P, γ_{opt} adds the k rules*

$$\mathbf{L}_D \text{ :- } \phi_1. \qquad \ldots \qquad \mathbf{L}_D \text{ :- } \phi_k.$$

Consequently, the corresponding DLP program of P is

$$\sigma_{opt}(P) = \bigcup_{r \in P} \sigma_{opt}(r).$$

Example 4.6. *Let consider the program P of the Example 4.4, $\sigma_{opt}(P)$ is the following program:*

$$\begin{array}{lll} \mathbf{L}_{a,b} \text{ :- } a, b. & a \text{ :- } \mathbf{L}_{a,b}. & b \text{ :- } \mathbf{L}_{a,b}. \\ \mathbf{L}_{c,d} \text{ :- } c, d. & c \text{ :- } \mathbf{L}_{c,d}. & d \text{ :- } \mathbf{L}_{c,d}. \end{array}$$

$$\begin{array}{ll} \mathbf{L}_{B(r_1)} \text{ :- } f. & \mathbf{L}_{B(r_1)} \text{ :- } g. \\ \mathbf{L}_{a,b} \vee \mathbf{L}_{c,d} \text{ :- } \mathbf{L}_{B(r_1)}. & f. \end{array}$$

Note that $|\gamma_{opt}(P)| < |\gamma(P)|$.

Theorem 4.2.2. *Let P be a ground NFN program on the alphabet Π and let $\sigma_{opt}(\Pi)$ be the corresponding DLP program built according to the Definition 4.5, then*

1. *the time required to compute $\sigma_{opt}(P)$ is polynomial in the size of P;*
2. *$AS(P) = \{I \cap \Pi \mid I \in AS(\sigma_{opt}(P))\}$;*
3. *$\sigma_{opt}(P \cup P_1) = \sigma_{opt}(P) \cup \sigma_{opt}(P_1)$, where P_1 is a ground NFN program.*

Proof. From the construction of the translation the items 3 and 1 follow. For the proof of the item 2 see the proof of the more general Proposition 4.3.5. □

Example 4.7. *Consider the program P and the corresponding DLP program $\sigma_{opt}(P)$ of the Example 4.6. Then, the answer sets of $\sigma_{opt}(P)$ are:*

$$AS'_1 = \{f, \mathbf{L}_{B(r_1)}, \mathbf{L}_{a,b}, a, b\}; \qquad AS'_2 = \{f, \mathbf{L}_{B(r_1)}, \mathbf{L}_{c,d}, c, d\}.$$

Therefore $AS'_1 \cap \Pi = \{f, a, b\}$ and $AS'_2 \cap \Pi = \{f, c, d\}$ are the answer sets of P.

4.3 Efficient Translation: Algorithm *rewriteNFN*

The translation described in Section 4.2 is an efficient translation from ground *NFN* programs to ground *DLP* programs based on structure-preserving normal form translations. However, it works only in the propositional case, in which variables cannot be used in problem representation. This restriction limits the suitability of nested logic programs in many application domains, especially when reasoning is to be done on large numbers of input facts.

In this section we describe the transformation algorithm *rewriteNFN*, which is efficient (similar to the efficient ground translation described in Section 4.2) and preserves safety and answer sets.

The algorithm *rewriteNFN* is based on structure-preserving transformations but needs several non-straightforward modifications in order to maintain safety. Since *rewriteNFN* introduces new symbols, the answer sets of the transformed program are not equal (as for the naïve transformation), but in a one-to-one correspondence to the answer sets of the original program (as for the efficient translation σ_{opt} described in Section 4.2.1), for which the original answer sets can be obtained by applying a simple projection.

The main structure of the algorithm is shown in Figure 4.1. It processes one rule at a time and cumulatively creates the transformed program P_{DLP}. For each *NFN* rule, a *DLP* one, called *major rule* H :- B, is created. The head of the major rule H is a disjunction comprised of as many disjuncts as there are in the original rule head, where non-atomic disjuncts have been replaced by new *label atoms* (and atomic disjuncts remain unaltered). Creating label atoms and their defining rules is handled by the procedure *buildAuxiliaryHeadAtom*, discussed in Section 4.3.3. The body of the major rule B is a conjunction comprised of as many conjuncts as the original rule body had, where non-atomic conjuncts have been replaced by label atoms (and atomic conjuncts remain unaltered). The handling of label atoms and their defining rules for rule bodies is more subtle (as safety issues need to be dealt with) and is done by the procedure *buildAuxiliaryBodyAtom*, described in Section 4.3.2. There are a few extra atoms and defining rules to be added for particular kinds of variable occurrences in the rule body, which are handled by the procedures *matchSafeShared*, *matchUnResShared*, and *defineDom*, which will also be discussed in Section 4.3.2.

4.3.1 Informal Overview

In this section we will examine the critical issues that *rewriteNFN* has to resolve. In particular we will motivate the handling of rule bodies.

Consider the rule

$$a(X) \text{ :- } (b(X,Y) \vee c(X)), (\textbf{not } d(X) \vee e(Y)). \tag{4.2}$$

Note that this rule is safe, even though the variable Y is unsafe—it does not occur in the rule head or a negative literal. Variable X, on the other hand, is safe.

begin *rewriteNFN*
Input: safe *NFN* program P
var B: conjunction of literals; H: disjunction of atoms;
Output: *DLP* program P_{DLP}.
  $P_{DLP} := \emptyset$;
  for each rule $r \in P$ **do**
    $H := \epsilon$; $B := \epsilon$;
    for each $C \in H(r)$ **do**
      if $H(r)$ contains only one atom a
        $H := H \vee a$;
      else
        buildAuxiliaryHeadAtom(C, H, P_{DLP}); (see Section 4.3.3)
      end if
    end for
    for each $D \in B(r)$ **do**
      if D contains only one literal l
        $B := B, l$;
      else
        buildAuxiliaryBodyAtom(D, B, P_{DLP}); (see Section 4.3.2)
      end if
    end for
    matchSafeShared(B, P_{DLP}); (see Section 4.3.2)
    matchUnResShared(B, P_{DLP}); (see Section 4.3.2)
    defineDom(P_{DLP}); (see Section 4.3.2)
    $P_{DLP} := P_{DLP} \cup \{H$:- $B.\}$; (see Section 4.3.4)
  end for
  return P_{DLP};
end.

Figure 4.1: Algorithm: *rewriteNFN*

If we lifted Definition 4.5 to the non-ground case without modifications we would obtain a translation as follows. We associate an auxiliary predicate aux^r_D, with all variables of D as arguments, to each basic disjunction $D \in B(r)$ and we would obtain the following program P_{un}

$$
\begin{aligned}
&r_{DLP} : a(X) \text{ :- } aux^r_{D_1}(X,Y), aux^r_{D_2}(X,Y).\\
&r1 : aux^r_{D_1}(X,Y) \text{ :- } b(X,Y).\\
&r2 : aux^r_{D_1}(X,Y) \text{ :- } c(X).\\
&r3 : aux^r_{D_2}(X,Y) \text{ :- } \textbf{not } d(X).\\
&r4 : aux^r_{D_2}(X,Y) \text{ :- } e(Y).
\end{aligned}
$$

which contains unsafe rules $r2$, $r3$, and $r4$. On the other hand, $aux^r_{D_1}$ and $aux^r_{D_2}$ have to be of arity 2, because the values for X and Y of the original

rule must match also in the rewritten program to preserve the one-to-one correspondence between answer sets.

Consider the following example to see why we cannot simply delete variables from auxiliary atoms which do not occur in their defining rule bodies.

Example 4.8. *Consider the program* P_1

$s : ok$:- $(b(X,Y) \vee c(X)), d(Y).$
$d(2).$
$b(1,1).$

It is easy to see that the only answer set of P_1 *is* $I = \{d(2), b(1,1)\}$ *and in particular that* I *does not contain* ok*. Suppose that we rewrite* P_1 *using a simple lifting of Definition 4.5 deleting unsafe variables from the corresponding DLP program, we obtain* P_1^{DLP}

$s_m : ok$:- $\text{aux}_D^s(X), d(Y)$
$s_a : \text{aux}_D^s(X)$:- $b(X,Y).$
$s_b : \text{aux}_D^s(X)$:- $c(X).$
$d(2).$
$b(1,1).$

Therefore, the answer set of P_1^{DLP} *is*

$$I^{DLP} = \{d(2), b(1,1), \text{aux}_D^s(1), ok\}$$

which does not contain ok*. Consequently,* $I^{DLP} \cap B_P = \{d(2), b(1,1), ok\}$ *which is not an answer set of the NFN program. The problem is that the major rule* s_m *of* s *has lost information about the bounding for the variable* Y *in* $D = b(X,Y) \vee c(X)$*.*

We need to take care of variables in *NFN* rules that occur in more than one basic disjunction of the same rule body (or in the rule head).

Definition 4.6 (Shared Variables)**.** *Given an NFN rule* r*, a variable* X *is shared in* r*, if it appears in two different basic disjunctions of* r *(body shared in* r*), or if* X *appears in both head and body of the rule.*

As we have observed above, shared variables may be safe or unsafe in a rule.

Definition 4.7 (Unrestricted and Safe Shared Variables)**.** *Let* D *a basic disjunction of an NFN rule* r*, then a variable* $X \in vars(D)$*, such that* X *is shared in* r*,*

- *is* safe shared *in D if X is a safe variable of r but X is not saved by D;*
- *is* unrestricted shared *in D if X is not a safe variable of r.*

A variable X is safe shared in r if it is safe shared in some basic disjunction of r, and X is unrestricted shared in r if it is unrestricted shared in some basic disjunction of r.

In rule (4.2), X is safe shared, while Y is unrestricted shared, because X is safe shared in (**not** $d(X) \vee e(Y)$), while Y is unrestricted shared in both basic disjunctions. Moreover, variable Y is safe shared in s of Example 4.8.

The basic idea used in our algorithm is to use a special, fresh constant $\#u$ instead of unsafe variables in rule heads defining auxiliary atoms for body literals. The constant $\#u$ can be read as *can take any value.* In our example P_{un}, the defining rules then become

$$r1: \ aux^r_{D_1}(X,Y) \text{ :- } b(X,Y).$$
$$r2: \ aux^r_{D_1}(X,\#u) \text{ :- } c(X).$$
$$r3: \ aux^r_{D_2}(X,\#u) \text{ :- } \textbf{not } d(X).$$
$$r4: \ aux^r_{D_2}(\#u,Y) \text{ :- } e(Y).$$

We observe now that $r3$ is still unsafe. This is because X occurs in a negative literal in D_2, but since rule (4.2) is safe, X must occur safe shared in D_2, implying that there is another basic disjunction (D_1 in this case) that saves X. So we add an atom with a new predicate symbol dom^r_X, which defines the possible values of variable X in r. Since X is safe all possible values are determined by the basic disjunctions that save X so we obtain the following rules

$$r3: \ aux^r_{D_2}(X,\#u) \text{ :- } \textbf{not } d(X), dom^r_X(X).$$
$$r5: \ dom^r_X(X) \text{ :- } aux^r_{D_1}(X,Y).$$

Now we have shown how to make the rules which define the auxiliary predicates safe. But we still have to deal correctly with the new constant $\#u$, which should be considered as being able to take an arbitrary value.

Example 4.9. *Consider the program P consisting only of rule (4.2) and the fact $b(1,1)$. The answer set of this program is $\{b(1,1), a(1)\}$.*

The rewritten program of P, as seen so far, is the following

r_{DLP} : $a(X)$:- $\text{aux}^r_{D_1}(X,Y), \text{aux}^r_{D_2}(X,Y)$.
$r1$: $\text{aux}^r_{D_1}(X,Y)$:- $b(X,Y)$.
$r2$: $\text{aux}^r_{D_1}(X,\#u)$:- $c(X)$.
$r3$: $\text{aux}^r_{D_2}(X,\#u)$:- ***not*** $d(X), \text{dom}^r_X(X)$.
$r4$: $\text{aux}^r_{D_2}(\#u,Y)$:- $e(Y)$.
$r5$: $\text{dom}^r_X(X)$:- $\text{aux}^r_{D_1}(X,Y)$.
$b(1,1)$.

The answer set of the DLP program is $J = \{\text{aux}^r_{D_1}(1,1), \text{dom}^r_X(1), \text{aux}^r_{D_2}(\#u,1), b(1,1)\}$ *and* $J \cap \Pi = \{b(1,1)\}$ *that is not an answer set of P. Indeed, rule r_{DLP} will not give rise to $a(1)$, because the constants 1 and $\#u$ do not match. Yet $\#u$ is supposed to mean "*can take any value*" and therefore should match any constant.*

We need to define the nonstandard behavior of $\#u$ explicitly. We do this for this example by defining predicates $match^r_X$ and $match^r_Y$ which define the matching for a particular variable in the presence of $\#u$. If the variable in question is unrestricted shared, in our case Y, we define $match^r_Y$ of arity 3, where the first two arguments are the values to be matched, while the third argument represents the matched value. If the first two arguments are both regular constants (i.e., not $\#u$), then they must be equal and in this case the third argument must be the same constant as the first two. If one of the first two arguments is $\#u$ and the other a regular constant, then the third argument is the regular constant. In this case $\#u$ can be thought of *having taken a value*. If the first two arguments are both $\#u$, then also the third argument is $\#u$ (can still take a value).

Therefore, if Y_1 and Y_2 are the new names for the variable Y appearing in the basic disjunction D_1 and D_2 respectively, Y the resulting match between Y_1 and Y_2, and c a constant, all the possible matches are

Y_1	Y_2	Y
$\#u$	$\#u$	$\#u$
$\#u$	c	c
c	$\#u$	c
c	c	c

To simulate this match we add to the *DLP* program the rules:

$r8: \ match^r_Y(Y,Y,Y) \ \text{:-} \ dom^r_Y(Y).$
$r9: \ match^r_Y(Y,\#u,Y) \ \text{:-} \ dom^r_Y(Y).$
$r10: \ match^r_Y(\#u,Y,Y) \ \text{:-} \ dom^r_Y(Y).$
$r11: \ match^r_Y(\#u,\#u,\#u).$

Since we also want to restrict the values these constants can take, we use the new predicate dom^r_Y for this purpose. dom^r_Y is slightly different to dom^r_X, as Y is an unrestricted shared variable, so there is no saving basic disjunction. We define it simply by means of each positive body literal in which it occurs

$r12: \ dom^r_Y(Y) \ \text{:-} \ b(X,Y).$
$r13: \ dom^r_Y(Y) \ \text{:-} \ e(Y).$

If the variable in question is safe shared, then at least one basic disjunction in the body is guaranteed to yield a value for this variable. X is a safe shared variable, and it will always obtain a value from D_1. We can thus optimize a bit with respect to unrestricted shared variables and define $match^r_X$ of arity 2, where the first argument will come from a basic disjunction that saves the variable. So the first argument will never be $\#u$, and the predicate will hold if the two arguments are the same constant, or if the second argument is $\#u$.

Therefore, if X is the safe variable appearing in D_1, X_2 is the new name of the safe shared variable X appearing in the basic disjunction D_2, and c a constant we can have only the following cases

X	X_2
c	$\#u$
c	c

We simulate this match adding the rules

$r6: \ match^r_X(X,X) \ \text{:-} \ dom^r_X(X).$
$r7: \ match^r_X(X,\#u) \ \text{:-} \ dom^r_X(X).$

In order to restrict the values these constants can take, we reuse the predicate dom^r_X.

As a final step, we modify r_{DLP} such that the modified matching becomes effective. This step involves renaming variable occurrences. For more than two occurrences, we would have to form a chain of *match* predicates:

$$r_{DLP}: \ a(X) \ \text{:-} \ aux^r_{D_1}(X,Y_1), aux^r_{D_2}(X_1,Y_2), \quad match^r_X(X,X_1), \ match^r_X(Y_1,Y_2,Y). \tag{4.3}$$

In this example we have informally seen all of the peculiar tasks that arise when translating rule bodies. In the following section, we will see how these techniques can be generalized.

4.3.2 Body Transformation

In this section we discuss the procedures *matchSafeShared*, *matchUnResShared*, *defineDom*, and *buildAuxiliaryBodyAtom*, which treat the rule bodies of the original *NFN* program. As seen in Figure 4.1, the algorithm *rewriteNFN* invokes *buildAuxiliaryBodyAtom* for each non-atomic basic disjunction, while procedures *matchSafeShared*, *matchUnResShared*, and *defineDom* are called once per rule.

In the rest of this section, we will assume that any set of variables can be turned unambiguously into a sequence (and viceversa) by using a fixed order over all variables.

buildAuxiliaryBodyAtom

The task of this procedure is to build an auxiliary atom representing the given basic disjunction and add it to the body of the major rule to be created. Moreover, it adds rules which define the auxiliary predicate. Let us define the atom which *buildAuxiliaryBodyAtom* adds to the body of the major rule.

Definition 4.8. *The procedure* buildAuxiliaryBodyAtom *given a basic disjunction D of an NFN rule r, creates an atom* $\mathrm{aux}^r_D(V_1, \ldots, V_n)$ *where* aux^r_D *is a new predicate symbol and $V_1, \ldots, V_n$ are the shared variables in* $\mathrm{vars}(D)$*, and adds it to B (that is the body of the corresponding standard rule of r built by* rewriteNFN*).*

For simplicity, we will often refer to $aux^r_D(V_1, \ldots, V_n)$ as aux^r_D in the sequel.

Example 4.10. *Consider the rule (4.2), for the basic disjunction $b(X, Y) \vee c(X)$ the atom* $\mathrm{aux}^r_{D_1}(X, Y)$ *is built while for the basic disjunction* ***not*** *$d(X) \vee e(Y)$ the atom* $\mathrm{aux}^r_{D_2}(X, Y)$ *is built. Therefore, the procedure adds the following atoms to B:*

$$\mathrm{aux}^r_{D_1}(X, Y), \mathrm{aux}^r_{D_2}(X, Y). \tag{4.4}$$

The defining rules, which *buildAuxiliaryBodyAtom* adds to P_{DLP}, can be characterized as follows:

Definition 4.9. *Given a basic disjunction D of an NFN rule r, the procedure* buildAuxiliaryBodyAtom *adds rules*

$$\{\text{aux}^r_D\sigma_{\#u} \text{ :- } a. \mid \textit{atom } a \textit{ in } D\} \tag{4.5}$$

$$\{\text{aux}^r_D\sigma_{\#u} \text{ :- } \textbf{\textit{not}}\ a, \text{dom}^r_{X_1}(X_1), \ldots, \text{dom}^r_{X_n}(X_n). \mid \textbf{\textit{not}}\ a \textit{ in } D, \text{vars}(a) = X_1, \ldots, X_n\} \tag{4.6}$$

to P_{DLP}, where, respectively, $\sigma_{\#u}$ is the substitution

$$\{v \mapsto \#u \mid v \in \text{vars}(\text{aux}^r_D) \setminus \text{vars}(a)\}.$$

As we illustrated in Section 4.3.1, predicates dom^r_X and the constant $\#u$ (which must not occur in the original *NFN* program) are necessary to create safe rules defining the auxiliary predicates.

Example 4.11. *Consider the rule (4.2), for the basic disjunction $b(X,Y) \vee c(X)$ the procedure* buildAuxiliaryBodyAtom *adds the following rules to P_{DLP}:*

$$r1: \text{aux}^r_{D_1}(X,Y) \text{ :- } b(X,Y).$$
$$r2: \text{aux}^r_{D_1}(X,\#u) \text{ :- } c(X).$$

Moreover, for the basic disjunction ***not*** *$d(X) \vee e(Y)$ it adds the rules:*

$$r3: \text{aux}^r_{D_2}(X,\#u) \text{ :- } \textbf{\textit{not}}\ d(X), \text{dom}^r_X(X).$$
$$r4: \text{aux}^r_{D_2}(\#u,Y) \text{ :- } e(Y).$$

to the standard program.

Defining rules for dom^r_X will be added later by the procedure *defineDom*.

matchSafeShared

This procedure is called once per rule, and its task is to handle the matching of the new constant $\#u$ for safe shared variables, as discussed in Section 4.3.1.

Definition 4.10. *Given an NFN rule r and a safe shared variable X in r,* matchSafeShared *adds the following two rules to P_{DLP}:*

$$\text{match}^r_X(X,X) \text{ :- } \text{dom}^r_X(X). \quad \text{match}^r_X(X,\#u) \text{ :- } \text{dom}^r_X(X). \tag{4.7}$$

Example 4.12. *For the rule (4.2), procedure* matchSafeShared *adds the following rules to* P_{DLP}*:*

$$r6: \text{match}^r_X(X, X) \text{ :- } \text{dom}^r_X(X).$$
$$r7: \text{match}^r_X(X, \#u) \text{ :- } \text{dom}^r_X(X).$$

Again, defining rules for the predicate dom^r_X will be added later by the procedure *defineDom*. *matchSafeShared* also changes the *DLP* rule body B, and replaces each occurrence of a safe shared variable in an auxiliary atom with a fresh one and then adds appropriate *match* atoms.

Definition 4.11. *Given an NFN rule* r*, a safe shared variable* X *in* r*, and a DLP rule body* B *that has been created by calls to* buildAuxiliaryBodyAtom, matchSafeShared *replaces each occurrence of* X *in an auxiliary atom* aux^r_D *by a new variable* X_D*, if* X *is safe shared in* D*. Moreover,* matchSafeShared *adds an atom* $\text{match}^r_X(X, X_D)$ *to* B *for each basic disjunction* D *in which* X *is safe shared.*

Example 4.13. *Consider the rule (4.2), the corresponding DLP rule body* B*, built so far, is (4.4). Then* matchSafeShared *modifies* B *obtaining:*

$$\text{aux}^r_{D_1}(X, Y), \text{aux}^r_{D_2}(X_1, Y), \text{match}^r_X(X, X_1). \tag{4.8}$$

matchUnResShared

This procedure is similar to *matchSafeShared*, but works on unrestricted shared variables. As discussed in Section 4.3.1, it defines *match* predicates with arity 3, and also adds *match* atoms to the body of the major rule in a slightly different way.

Definition 4.12. *Given an NFN rule* r *and an unrestricted shared variable* X *in* r*,* matchUnResShared *adds the following rules to* P_{DLP}*:*

$$\text{match}^r_X(X, X, X) \text{ :- } \text{dom}^r_X(X). \qquad \text{match}^r_X(\#u, X, X) \text{ :- } \text{dom}^r_X(X). \tag{4.9}$$
$$\text{match}^r_X(X, \#u, X) \text{ :- } \text{dom}^r_X(X). \qquad \text{match}^r_X(\#u, \#u, \#u). \tag{4.10}$$

As mentioned earlier, the predicate dom^r_X will be defined later in procedure *defineDom*.

Example 4.14. *Consider the rule (4.2), procedure* matchUnResShared *adds the following rules to* P_{DLP}

$$r8: \text{match}^r_Y(Y,Y,Y) \text{ :- } \text{dom}^r_Y(Y).$$
$$r9: \text{match}^r_Y(Y,\#u,Y) \text{ :- } \text{dom}^r_Y(Y).$$
$$r10: \text{match}^r_Y(\#u,Y,Y) \text{ :- } \text{dom}^r_Y(Y).$$
$$r11: \text{match}^r_Y(\#u,\#u,\#u).$$

Just like *matchSafeShared*, *matchUnResShared* also changes the *DLP* rule body B, and replaces each occurrence of an unrestricted shared variable in an auxiliary atom with a fresh one and then adds appropriate *match* atoms. The latter is somewhat more involved, as we need to form a chain of matches, eventually ending with the original variable.

Definition 4.13. *Given an NFN rule* r*, an unrestricted shared variable* X *in* r*, and a DLP rule body* B *that has been created by calls to* buildAuxiliaryBodyAtom, matchUnResShared *replaces each occurrence of* X *in an auxiliary atom* aux^r_D *by a new variable* X_D*. Let* $X_1, \ldots, X_n$ *be a sequence of the renamed variables,* matchUnResShared *adds the following atoms to* B $\text{match}^r_X(X_1, X_2, X_{\leq 2})$*, for each* $2 \leq i < n-1$ $\text{match}^r_X(X_{\leq i}, X_{i+1}, X_{\leq i+1})$*, and finally* $\text{match}^r_X(X_{\leq n-1}, X_n, X)$ *to* B *if* $n > 2$*. For the special case* $n = 2$*, just* $\text{match}^r_X(X_1, X_2, X)$ *is added to* B*.*

Example 4.15. *Consider the rule (4.2), the corresponding DLP rule body* B*, built so far, is (4.8). Then* matchUnResShared *modifies* B *obtaining:*

$$\text{aux}^r_{D_1}(X, Y_1), \text{aux}^r_{D_2}(X_1, Y_2), \text{match}^r_X(X, X_1), \text{match}^r_X(Y_1, Y_2, Y). \quad (4.11)$$

After the application of this procedure the building of the body of the major rule is completed.

defineDom

This procedure adds definitions for the *dom* predicates. These definitions are needed only for safe and unrestricted shared variables in r. The definition for safe shared variables is simpler, it can re-use the *aux* predicate of a basic disjunction that saves the variable in question. For unrestricted shared variables, we use all body literals of the original NFN rule in which the variable occurs. Note that, since an unrestricted shared variable is not safe, it cannot occur in any negative literal if the original rule is safe.

Definition 4.14. *Given an NFN rule r and a safe shared variable X in r,* defineDom *adds a rule*

$$\mathrm{dom}^r_X(X) \text{ :- } \mathrm{aux}^r_{D_i}. \tag{4.12}$$

where D_i is some basic disjunction that saves X. Without loss of generality, we choose $i = min\{j \mid D_j \, saves \, X, \, D_j \in B(r)\}$.

Given an unrestricted shared variable X in r, defineDom *adds a rule*

$$\mathrm{dom}^r_X(X) \text{ :- } l. \tag{4.13}$$

for each literal $l \in B(r)$ in which X occurs.

Example 4.16. *Consider the rule (4.2) where X is a safe shared variable and Y is an unrestricted shared variable, then for the variable X procedure* defineDom *adds only the rule*

$$r5 : \mathrm{dom}^r_X(X) \text{ :- } aux^r_{D_1}(X, Y).$$

while for the variable Y it adds two rules

$$r12 : \mathrm{dom}^r_Y(Y) \text{ :- } b(X, Y).$$
$$r13 : \mathrm{dom}^r_Y(Y) \text{ :- } e(Y).$$

4.3.3 Head Transformation

Rewriting an *NFN* rule head is simpler then rewriting an *NFN* rule body, because we do not have to deal with safety issues, as each variable occurring in the head needs to be safe. Moreover, each head variable is a shared variable, and therefore will occur in the rule body of the transformed major rule. As a consequence, the rewriting of the rule head is a fairly direct lifting of the respective transformation in Definition 4.5. Obviously, in our case we have to take care of variables. As a consequence, *buildAuxiliaryHeadAtom* is essentially a simplification of *buildAuxiliaryBodyAtom*.

Definition 4.15. *Given a basic conjunction C of an NFN rule r, the procedure* buildAuxiliaryBodyAtom *creates an atom* $\mathrm{auxh}^r_C(V_1, \ldots, V_n)$ *where* auxh^r_C *is a new predicate symbol and* $\mathrm{vars}(C) = \{V_1, \ldots, V_n\}$*, and adds it to H.*

For simplicity, we will often refer to $auxh^r_C(V_1, \ldots, V_n)$ as $auxh^r_C$ in the sequel.

Example 4.17. *Consider the NFN rule*

$$r : a(X), b(Y) \vee c, d(Z), f(X) \text{ :- } e(X, Y, Z). \tag{4.14}$$

Therefore, buildAuxiliaryHeadAtom *builds the atom* $\text{auxh}^r_{C_1}(X, Y)$ *for the basic conjunction* $a(X), b(Y)$ *and it creates the atom* $\text{auxh}^r_{C_2}(X, Z)$ *for the basic conjunction* $c, d(Z), f(X)$*. Consequently, the DLP rule head* H *is*

$$\text{auxh}^r_{C_1}(X, Y) \vee \text{auxh}^r_{C_2}(X, Z).$$

In addition to rules defining $auxh^r_C$ (as for auxiliary predicates in the rule body), we also need rules that ensure the truth of all atoms in a basic conjunction if its auxiliary atom becomes true.

Definition 4.16. *Given a basic conjunction* C *of an NFN rule* r*, the procedure* buildAuxiliaryHeadAtom *adds the rule*

$$\text{auxh}^r_C \text{ :- } C. \tag{4.15}$$

and for each atom a *in* C *a rule*

$$a \text{ :- auxh}^r_C. \tag{4.16}$$

Example 4.18. *Consider the rule (4.14) in the previous example, then the corresponding DLP program of* r *is the following*

$$\begin{array}{l}
\text{auxh}^r_{C_1}(X, Y) \text{ :- } a(X), b(Y). \\
\text{auxh}^r_{C_2}(X, Z) \text{ :- } c, d(Z), f(X). \\
a(X) \text{ :- auxh}^r_{C_1}(X, Y). \\
b(Y) \text{ :- auxh}^r_{C_1}(X, Y). \\
d(Z) \text{ :- auxh}^r_{C_2}(X, Z). \\
f(X) \text{ :- auxh}^r_{C_2}(X, Z). \\
c \text{ :- auxh}^r_{C_2}(X, Z).
\end{array}$$

$$r_m : \text{auxh}^r_{C_1}(X, Y) \vee \text{auxh}^r_{C_2}(X, Z) \text{ :- } e(X, Y, Z). \tag{4.17}$$

In order to see why rules (4.15) are needed, consider the following (ground) example.

Example 4.19. *Let* P

$$\begin{array}{l}
a \vee b, c. \\
b. \\
c.
\end{array}$$

The transformed program P_{DLP} without rules generated by (4.15) would be

$a \vee$ auxh.
b :- auxh.
c :- auxh.
b.
c.

The answer sets of P_{DLP} are $AS = \{\{\text{auxh}, b, c\}, \{a, b, c\}\}$, while P has only an answer set $\{b, c\}$.

4.3.4 Major Rule

For each *NFN* rule, the algorithm *rewriteNFN* builds a corresponding standard rule, called *major rule* that directly represents the *NFN* rule in the standard program. The head of the major rule is built by *buildAuxiliaryHeadAtom* and the body by *buildAuxiliaryBodyAtom*. Putting head and body together, the algorithm *rewriteNFN* adds the major rule as the final step to the standard program.

In detail, if r is the following rule

$$C_1 \vee \cdots \vee C_n \text{ :- } D_1, \ldots, D_m.$$

the corresponding major rule is of the form:

$$auxh^r_{C_1} \vee \cdots \vee auxh^r_{C_n} \text{ :- } aux^r_{D_1}, \ldots, aux^r_{D_m}, MATCH^r_{X_i}, \ldots, MATCH^r_{X_k}. \quad (4.18)$$

where $X_1, \ldots, X_k$ are shared variables and $MATCH^r_{X_i}$ represents the conjunction of the *match* atoms added by functions *matchSafeShared* or *matchUnResShared* for the variable X_i.

Example 4.20. *The rule (4.3) is the corresponding major rule built for the rule (4.2). Moreover, the rule (4.17) is the corresponding major rule for the NFN rule (4.14).*

4.3.5 Properties of the Algorithm

In this section we show some important properties of algorithm *rewriteNFN*, namely that it maintains safety, that the answer sets of the transformed program are in a one-to-one correspondence to the answer sets of the original program, and that the obtained program is of polynomial size.

We will first consider safety. In the algorithm we had to take a lot of care for ensuring safety of the generated rules. We will now formally prove that safety is guaranteed.

Proposition 4.3.1. *Let P be a safe NFN program, then P_{DLP} generated by* rewriteNFN *is safe.*

Proof. Observe that we can partition the rules of P_{DLP} by the rule in P that gave rise to their generation. Let P_r be the set of rules in P_{DLP} created for a rule $r \in P$. We show that safety of r implies safety of P_r.

To do this we will consider all rules that are generated for r. Rules of the form (4.5) and (4.6) are safe because the substitution $\sigma_{\#u}$ ensures that each variable in the head appears also in the body; moreover, in a rule of the form (4.6) each variable V in a negative literal appears also in a positive body atom $dom^r_V(V)$, while in a rule of the form (4.5) there are no negative literals. Rules of the form (4.7), (4.9), and (4.10) are trivially safe. Rules of the form (4.12) are safe as the head variable occurs in the positive body by construction. For rules of the form (4.13) we observe that unrestricted shared variables may not occur in negative literals of r because these variables are not safe, so rules (4.13) are guaranteed to have a positive body containing the head variable. Rules of the form (4.15) and (4.16) are safe by construction, as aux^r_C contains all variables of C.

Finally, consider the major rule r' generated in place of r. Note that all variables in the head of r are shared variables, and thus they must occur in some body literal of r'. Moreover, since r is safe, then each head variable must occur in a positive basic conjunction, causing the variable to occur in a positive body literal of r'. The only negative literals that r' may contain stem from basic disjunctions in r containing exactly one negative literal. So variables occurring in negative literals in r' need to occur in a positive basic disjunction of r, which gave rise to a positive body literal in r', containing the variable. Given that *rewriteNFN* creates no other rules, we can conclude the proof. □

By the construction of the program *rewriteNFN* the following proposition follows immediately.

Proposition 4.3.2. *Let P_1 and P_2 be NFN programs then*

$$\text{rewriteNFN}(P_1 \cup P_2) = \text{rewriteNFN}(P_1) \cup \text{rewriteNFN}(P_2).$$

We next show that there is a one-to-one correspondence between answer sets of a safe *NFN* program P and answer sets of the program produced by *rewriteNFN*(P).

Definition 4.17. *Given a safe NFN program P and I an interpretation for P, we define its expansion as :*

$$I_D := I \cup \bigcup_{r \in P} (\pi_b^+(r) \cup \omega_s(r) \cup \omega_u(r) \cup \mu_s(r) \cup \mu_u(r) \cup \pi_b^-(r) \cup \pi_h(r))$$

where

$$\begin{aligned}
\pi_b^+(r) &= \{\mathrm{aux}_D^r\sigma' | \textit{atom } a \in D, D \in B(r), a\sigma \in I, \\
&\quad \sigma' = \{v \mapsto \sigma(v) | v \in \mathrm{vars}(a)\} \cup \{v \mapsto \#u | v \in \mathrm{vars}(r) \setminus \mathrm{vars}(a)\}\} \\
\omega_s(r) &= \{\mathrm{dom}_X^r(\sigma(X)) | X \textit{ safe shared in } r, D_i \textit{ the first basic disjunction} \\
&\quad \textit{saving } X, \mathrm{aux}_{D_i}^r\sigma \in \pi_b^+(r)\} \\
\omega_u(r) &= \{\mathrm{dom}_X^r(\sigma(X)) | X \textit{ unrestricted shared in } r, X \textit{ occurs in } l \in B(r), \\
&\quad l\sigma \in I\} \\
\mu_s(r) &= \{\mathrm{match}_X^r(c,c), \mathrm{match}_X^r(c,\#u) | X \textit{ safe shared in } r, \mathrm{dom}_X^r(c) \in \omega_s(r)\} \\
\mu_u(r) &= \{\mathrm{match}_X^r(c,c,c), \mathrm{match}_X^r(c,\#u,c), \mathrm{match}_X^r(\#u,c,c) | X \textit{ unrestricted} \\
&\quad \textit{shared in } r, \mathrm{dom}_X^r(c) \in \omega_u(r)\} \cup \{\mathrm{match}_X^r(\#u,\#u,\#u)\} \\
\pi_b^-(r) &= \{\mathrm{aux}_D^r\sigma' | \textbf{\textit{not}}\ a \in D, D \in B(r), a\sigma \notin I, \forall X \in \mathrm{vars}(a) : \\
&\quad \mathrm{dom}_X^r(\sigma(X)) \in \omega_s(r), \\
&\quad \sigma' = \{v \mapsto \sigma(v) | v \in \mathrm{vars}(a)\} \cup \{v \mapsto \#u | v \in \mathrm{vars}(r) \setminus \mathrm{vars}(a)\}\} \\
\pi_h(r) &= \{\mathrm{auxh}_C^r\sigma | C \in H(r), \forall a \in C : a\sigma \in I\}
\end{aligned}$$

Note that there is exactly one I_D for a fixed I.

Lemma 4.3.3. *Let P be an NFN program, I be an answer set for P, and I_D be the set built according to Definition 4.17 then I_D is a model for* rewriteNFN(P).

Proof. First of all, note that

$$I_D \cap B_P = I. \tag{4.19}$$

Moreover, for each rule $r \in P$ the algorithm *rewriteNFN* builds a corresponding standard program P_r coinciding with *rewriteNFN*(r) where $r_m \in P_r$

is the corresponding major rule of r. For each ground instance $r\sigma$ of r the corresponding ground standard program $P_r\sigma$ is

$$P_r\sigma = \{r_s\sigma \mid r_s \in P_r, r_s \neq r_m\} \cup \{r_m\sigma' \mid r_m\sigma' \in Ground(r_m, U_P \cup \#u), \sigma'_{|vars(r)} = \sigma\}.$$

Therefore, to show that I_D is a model for $P_{DLP} = rewriteNFN(P)$ it is enough to show that $I_D \models P_r\sigma$, for each $r\sigma \in Ground(P)$.

Let $r_s\sigma \in P_r\sigma$ be a rule such that $I_D \models B(r_s\sigma)$ and suppose that r_s was built by means of function *buildAuxiliaryBodyAtom*.

- If the rule is of the form (4.5), then $B(r_s\sigma) = a\sigma$ and $a\sigma \in D\sigma$. Since $I_D \models a\sigma$, from (4.19) $a\sigma \in I$. Consequently, from the construction of I_D, the auxiliary atom $aux^r_D\sigma' \in \pi^+_b(r)$ (that is $aux^r_D\sigma' \in I_D$). Moreover, note that $H(r_s\sigma) = aux^r_D\sigma'$, therefore $I_D \models r_s\sigma$.

- If the rule is of the form (4.6), since $I_D \models B(r_s\sigma)$, $I_D \models$ **not** $a\sigma$ then $a\sigma \notin I$ and for each $X \in vars(a)$, atom $dom^r_X\sigma \in I_D$. Therefore, from the construction of I_D, $aux^r_D\sigma' \in I_D$. Moreover, $H(r_s\sigma) = aux^r_D\sigma'$, and $I_D \models r_s\sigma$.

Similarly, it is no hard to see that for each rule $r_s\sigma$ built by procedures *matchSafeShared*, *matchUnResShared*, and *buildAuxiliaryHeadAtom*, $I_D \models r_s\sigma$.

So far we proved that for each basic disjunction $D\sigma \in B(r\sigma)$ such that $I \models D\sigma$, there is at least one atom $aux^r_D\sigma' \in I_D$, and for each basic conjunction $C \in H(r)$ such that $I \models C$ there is exactly one $auxh^r_C\sigma \in I_D$.

Finally, consider the major rule $r_m\sigma' \in P_r\sigma$ and suppose that $I_D \models B(r_m\sigma')$. Therefore, $aux^r_D\sigma' \in I_D$, for each $D \in B(r)$ then $I \models D\sigma$ for each $D\sigma \in B(r\sigma)$. Moreover, from the hypothesis that $I \in AS(P)$, $I \models H(r\sigma)$. Consequently, there exists a basic conjunction $C\sigma \in H(r\sigma)$ such that $I \models C\sigma$ and from definition of I_D, $auxh^r_C\sigma \in \pi_h(r)$. As a result, $I_D \models H(r_m\sigma')$, since from the construction, $H(r_m)$ is the disjunction of all auxiliary atom corresponding to the basic conjunctions of $H(r)$. □

Lemma 4.3.4. *Let P be an NFN program, I be an answer set for P, and I_D be the set built according to Definition 4.17, then*

$$I_D \in AS(\text{rewriteNFN}(P)).$$

Proof. From the Lemma 4.3.3, I_D is a model for $P_{DLP} = rewriteNFN(P)$. Assume that I_D is not a minimal model for $Ground(P_{DLP})^{I_D}$ then there exists a set of atoms $J_D \subsetneq I_D$ such that J_D is a model for $Ground(P_{DLP})^{I_D}$. Let J be the set $J_D \cap B_P$, we then prove that $J \subsetneq I$ and $J \models Ground(P)^I$ contradicting the hypothesis $I \in AS(P)$.

- $J \subsetneq I$
  From the hypothesis $J_D \subsetneq I_D$, there exists an atom $a_D^g \in I_D$ such that $a_D^g \notin J_D$. If $a_D^g = aux_D^{rg}$ then there exists a rule $r_s^g \in Ground(P_{DLP})^{I_D}$ of the form (4.5) or (4.6) such that $H(r_s) = aux_D^{rg}$ and $J_D \not\models B(r_s^g)$ (note that from the construction of I_D, since $aux_D^{rg} \in I_D$, $I_D \models B(r_s^g)$ and $r_s^g \in Ground(P_{DLP})^{I_D}$). If r_s^g is of the form (4.5), $B(r_s^g) = a^g$, where $a^g \in I$ (since $aux_D^{rg} \in I_D$), and $a^g \notin J$. Otherwise, if r_s^g is of the form (4.6), since $I_D \models \mathbf{not}\ a^g$ and $J_D \subsetneq I_D$, in order to satisfy $J_D \not\models B(r_s^g)$ an atom $dom_X^{rg} \in B(r_s^g)$ exists such that $dom_X^{rg} \notin J_D$. If dom_X^{rg} is the head of rule r_s^g of the form (4.12) an atom $aux_{D'}^{rg} \notin J_D$ where D' is a positive basic disjunction and from previous considerations an atom $a^g \in I$ exists such that $a^g \notin J$. Even if dom_X^{rg} is the head of a rule of the form (4.13), $B(r_s^g) = a^g$ and $a^g \in I$ (since $dom_X^{rg} \in I_D$) but $a^g \notin J$. Finally, if $a_D^g = match_X^{rg}$, defined by rule of the form (4.7), (4.9) or (4.10), an atom $dom_X^{rg} \notin J_D$ and, from previous considerations, there exists an atom a^g such that $a^g \in I$ and $a^g \notin J$. Consequently, $J \subsetneq I$.

- $J \models Ground(P)^I$
  Let $r\sigma \in Ground(P)$ be a ground *NFN* rule and $r_m\sigma'$ be a corresponding major rule. If $I_D \models B(r_m\sigma')$, then for each basic disjunction $D\sigma \in B(r\sigma)$ there is a corresponding auxiliary atom $aux_D^r\sigma' \in I_D$. Consequently, from the construction of I_D, there exists a literal $l\sigma \in D\sigma$ such that $I \models l\sigma$ for each $D\sigma \in B(r\sigma)$. As a result, $r_m\sigma' \in Ground(P_{DLP})^{I_D}$ if and only if $r\sigma \in Ground(P)^I$ (note that $r\sigma \in Ground(P)^I$ can have fewer literals in the body than $r\sigma \in Ground(P)$ but this is not relevant for the proof).

  If $J_D \models B(r_m\sigma')$ then from the construction of I_D, $J \models B(r\sigma)$ and, since $J_D \models H(r_m\sigma')$, $J \models H(r\sigma)$. Moreover, it is easy to see that if $J_D \not\models B(r_m\sigma')$ for each $r_m\sigma' \in Ground(P_{DLP})_D^I$ such that $\sigma'_{|vars(r)} = \sigma$, $J \not\models B(r\sigma)$. Consequently, J is a model for $Ground(P)^I$ and this is a contradiction to the hypothesis $I \in AS(P)$.

□

Proposition 4.3.5. *Let P be a safe NFN program, $P_{DLP} = \text{rewriteNFN}(P)$, then*

$$AS(P_{DLP}) = \{I_D \mid I \in AS(P)\}$$

.

Proof. Let I_D be the set of atoms built according to Definition 4.17 then, from Lemma 4.3.4, $I_D \in AS(P_{DLP})$.

Assuming that $K \in AS(P_{DLP})$, we can show that its restriction to the language of P, $K' = K \cap B_P$ is in $AS(P)$ and that $K'_D = K$. Again the major observation is that each major rule in $Ground(P_{DLP})$ corresponds to exactly one rule in $Ground(P)$. Moreover, an auxiliary atom representing a ground basic disjunction D is true in K only if some literal of D is true in K' due to construction and safety. Atoms in B_P however can only be true in K if the body of a ground rule stemming from rules (4.16) or an atom directly contained in a major rule (of an atomic basic conjunction in P) is satisfied by K. Rules (4.16) in turn can support these atoms only if there is a major rule that supports the respective auxiliary atoms. This however means that also the rule in P that corresponds to the major rule supports the same atoms, so K' must be a model of P. We can also check in a similar way that no $J \subsetneq K'$ satisfies the reduct of P. Finally, we are able to show that K must contain $(\pi_b^+(r) \cup \omega_s(r) \cup \omega_u(r) \cup \mu_s(r) \cup \mu_u(r) \cup \pi_b^-(r) \cup \pi_h(r))$ for K' and each rule $r \in P$ in order to satisfy P_{DLP}. This implies that K is of the form K'_D. □

Proposition 4.3.6. *Let P be a safe NFN program, then the output of the algorithm* rewriteNFN(P) *has polynomial size in $|P|$ and* rewriteNFN(P) *runs in polynomial time in $|P|$.*

Proof. Let m be the total number of rules in P, let k be the maximum number of variables appearing in a rule and let t be the maximum number of distinct predicates appearing in a rule. Let $n = \max\{n_d, n_c\}$ where n_d is the maximum number of basic disjunctions and n_c is the maximum number of basic conjunctions occurring in a rule. Moreover, let $j = \max\{j_d, j_c\}$ where j_d is the maximum number of literals appearing in a basic disjunction and j_c is the maximum number of atoms appearing in a basic conjunction.

For each rule $r \in P$ *rewriteNFN* adds to P_{DLP} at most $n * j$ rules of type (4.16), at most n of type (4.15), and at most $n * j$ of types (4.5) and (4.6). If all variables of a rule are unrestricted shared then *rewriteNFN* adds $k * t$

rules of type (4.13) and $k * 4$ of type (4.9) and (4.10). In all other cases (not all variables are unrestricted shared) we obtain fewer rules, possibly of different types. Consequently $|P_{DLP}| \leq m * [n(j+1) + (n * j + k(t+4))]$ where $|P_{DLP}|$ refers to the cardinality of P_{DLP}. Moreover, each rule in P_{DLP} clearly has polynomial size in $|P|$.

Next, observe that procedure *buildAuxiliaryHeadAtom* takes $O(j)$ steps, while *buildAuxiliaryBodyAtom* takes $O(j * k^2)$ steps, as for each literal in a basic disjunction, *buildAuxiliaryBodyAtom* iterates on each of its variables. Procedures *matchSafeShared* and *matchUnResShared* both need $O(n * j * k)$ steps while *defineDom* runs in $O(n*k)$. The first loop in *rewriteNFN* therefore runs in $O(n*j)$, and the second one invokes *buildAuxiliaryBodyAtom* n times, so is $O(n * j * k^2)$, which is also the bound for the processing of each rule. In total, *rewriteNFN* runs in $O(m * n * j * k^2)$. □

Chapter 5

Systems `nfn2dlp` and `nfnsolve`

In this chapter we describe the tools `nfn2dlp` and `nfnsolve`. The former implements the algorithm *rewriteNFN* described in Section 4.3, the latter computes the answer sets of *NFN* programs.

The chapter is organized as follows:

- ★ In Section 5.1 we describe the implementation of `nfn2dlp` and also the commandline interface is described.
- ★ In Section 5.2 we describe the implementation of the tool `nfnsolve` and its interface.

class TERM
  `attr :term`
methods
  `isConst` *returns true if* `term` *is a constant*
  `isAnonVar` *returns true if* `term` *is the string* `_`
  `isVar` *returns true if* `term` *is a variable*

Figure 5.1: class TERM

5.1 The tool `nfn2dlp`

The system `nfn2dlp` translates safe *NFN* programs into safe *DLP* programs. It provides an *NFN* parser and safety checker, and an implementation of the translation *rewriteNFN* presented in Section 4.3. The output program is in the format of DLV, state-of-the-art implementation for disjunctive logic programs under the answer set semantics, and thus allows for effective answer set computation of *NFN* programs.

5.1.1 Implementation of `nfn2dlp`

The tool `nfn2dlp` has been implemented using the language Ruby [8]. Ruby is object-oriented: Every data type is an object, including classes and types which many other languages designate as primitives (such as integers, booleans, and "nil"). This language supports multiple programming paradigms, including functional, object oriented, imperative and reflective. It also has a dynamic type system and automatic memory management.

Classes for Language Constructs

The tool `nfn2dlp` relies on a code base which has been constructed using an object-oriented design. For each language construct we implemented an appropriate Ruby class described in the following. All classes contain methods set and get. Moreover, they are represented by pseudocode.

First of all, we define the class TERM (see Figure 5.1) representing a term. The attribute of this class is a string according to Definition 1.4. Moreover, it includes methods identifying whether a term is a constant or a variable or an anonymous variable (the string "_").

The class ATOM (see Figure 5.2) represents atoms, according to Definition 1.5. The class LITERAL (see Figure 5.3), representing literals, is defined

class ATOM
  `attr :predicate_name`
  `attr :terms` *represents the list of terms appearing in the atom and can be empthy*

Figure 5.2: class ATOM

class LITERAL
  `attr :atom`
  `attr :isNeg` *is a boolean variable that is true if the literal is negative, false otherwise*

Figure 5.3: class LITERAL

according to Definition 1.6. For the class ATOM and LITERAL we provided also all methods to manipulate the constructs in the other classes of the tool.

Using classes ATOM and LITERAL we defined classes BASIC-CONJUNCTION (see Figure 5.4) and BASIC-DISJUNCTION (see Figure 5.5) representing constructs basic conjunction (see Definition 2.2) and basic disjunction (see Definition 2.1), respectively.

The attributes in the class BASIC-CONJUNCTION are the atoms and the set of variables appearing in a basic conjunction. The set of variables is computed by function `set_vars` (accessing the set of terms in the class ATOM).

class BASIC-CONJUNCTION
  `attr :atoms`
  `attr :vars` *contains all variables appearing in* `atoms`
methods
  `set_vars` *computes the set* `vars`

Figure 5.4: class BASIC-CONJUNCTION

In the class BASIC-DISJUNCTION the attributes are the literals appearing in a basic disjunction and also the sets of its safe, safe shared and unrestricted shared variables. To compute the set of safe shared and unrestricted shared variables of a basic disjunction we need access to the body containing the basic disjunction. Therefore, the methods `set_sharedVars(body)` (see Figure 5.6) and `set_unResVars(body)` (see Figure 5.7) are provided.

The class which represents the head of a rule is called DISJUNCTION (see Figure 5.8) and its attribute is a collection of basic conjunction. The method `vars` computes all variables appearing in the disjunction using the attribute

class BASIC-DISJUNCTION
  `attr :literals`
  `attr :allVars` *is the set of all variables*
  `attr :safeVars` *is the set of all safe variables*
  `attr :sharedVars` *is the set of all safe shared variables*
  `attr :unResVars` *is the set of all unrestricted shared variables*
methods:
  `set_sharedVars(body)`
  `set_unResVars(body)`

Figure 5.5: class BASIC-DISJUNCTION

```
def set_sharedVars(body)
 vars = allVars\safeVars
 sharedVars = ∅
 for each basic disjunction bd in body
  sharedVars = sharedVars ∪ (vars ∩ bd.safe_vars)
 end
end
```

Figure 5.6: method **set_sharedVars** of the class BASIC-DISJUNCTION

`vars` of the class BASIC-CONJUNCTION.

The class representing the body is called CONJUNCTION (see Figure 5.9) and its attribute is a collection of its basic disjunctions. This class provides methods `safeVars`, `safeShardVars`, `safeUnResVars` computing the set of safe, safe shared and unrestricted shared variables appearing in the body. All methods use the corresponding set of variables present in BASIC-DISJUNCTION objects.

We have also implemented the class RULE (see Figure 5.10) representing rules according to Definition 2.3.

The most important method of this class is the method `isSafe` (see Figure 5.11). The method checks if any atom in the head or any negative literal of the body contains any anonymous variable, in this case the rule is unsafe. Moreover, the method checks if all variables in the head and in the negative body literals are safe in the body of the rule.

Finally, we provide the class PROGRAM that represents programs and its attribute is a set of rules.

```
def set_unResVars(body)
 vars = allVars\safeVars
 unResVars = ∅
 for each basic disjunction bd in body
  unResVars = unResVars ∪ (vars ∩ bd.allVars)
 end
end
```

Figure 5.7: method **set_unResVars** of the class BASIC-DISJUNCTION

class DISJUNCTION
`attr :basic-conjunctions`
methods
`vars` *computes all variables occurring in* `basic-conjunctions`

Figure 5.8: class DISJUNCTION

class CONJUNCTION
`attr :basic-disjunctions`
methods

`negativeVars`	*computes negative literal variables*
`safeVars`	*computes all safe variables*
`safeShardVars`	*computes all safe shared variables*
`safeUnResVars`	*computes all unrestricted shared variables*
`includeNegLitAnonVars`	*returns true if any negative literal contains anonymous variables*

Figure 5.9: class CONJUNCTION

Parser

The *NFN* parser is implemented using the tool `treetop` [23]. It provides a parser generator for Parsing Expression Grammars (PEGs) [9] for Ruby. PEGs are a novel concept for parser specification, which look similar to classical BNF grammars but differ in semantics; most importantly these grammars avoid ambiguity.

The parser contains a collection of rules for recognizing the constructs. Moreover, during the parsing, the objects representing the language construct are built.

class RULE
  `attr :head`
  `attr :body`
methods
  `isSafe` *returns true is the rule is safe*

Figure 5.10: class RULE

```
begin isSafe
  for each basic conjunction bc in the head
    if bc contains anonymous variables
      return false
    end
  end
  if CONJUNCTION.includeNegLitAnonVars is true
    return false
  end
  safeHead_vars = DISJUNCTION.vars ∩ CONJUNCTION.safeVars
  safeNeg_vars = DISJUNCTION.negativeVars ∩ DISJUNCTION.safeVars
  if(safeHead_vars == CONJUNCTION.vars) &&
       (safeNeg_vars == CONJUNCTION.negativeVars)
    return true
  end
  return false
end.
```

Figure 5.11: method **isSafe** of the class RULE

Rewriting

Two classes for handling the rewriting have been defined, REWRITEHEAD and REWRITEBODY. For the full encoding of this classes, see Appendix B.

The class REWRITEHEAD contains as attribute `nfn_head` representing the head of *NFN* head, `dlp_head` representing the head of the corresponding *DLP* major rule and a set of auxiliary *DLP* rules. The *DLP* head `dlp_head` is built by means of function `rewriteHead`. This function is the effective implementation of the procedure *buildAuxiliaryHeadAtom* described in Section 4.3.3. The auxiliary *DLP* rules built by `rewriteHead` are collected in the attribute `dlp_rules`.

class REWRITERHEAD
  `attr :nfn_head`
  `attr :dlp_head`
  `attr :dlp_rules`
methods
  `rewriteHead`

The class REWRITEBODY contains attributes `nfn_body` representing the body of the *NFN* rule, `dlp_body` representing the body of the corresponding *DLP* major rule and `dlp_rules`, that is a set of auxiliary standard rules. It has also attributes `Dom`, a set of *dom* atoms, and `Match`, a set of *match* atoms. The method `rewriteBody` is the implementation of the procedure *buildAuxiliaryBodyAtom* described in Section 4.3.2. In particular, `buildAux` is used for building the corresponding auxiliary atom for each basic conjunction in `nfn_body`, `buildAtomMatch` constructs the set `Match`, and `buildDomAtom` builds the set of *dom* atoms. Moreover, `addDomRules` and `addMatchRules` create and add to `dlp_rules` the defining rules for the atoms in `Dom` and `Match`, respectively.

class REWRITERBODY
  `attr :nfn_body`
  `attr :dlp_body`
  `attr :dlp_rules`
  `attr :Dom`
  `attr :Match`
methods

`rewriteBody`	
`buildAux`	*builds* aux *atoms*
`buildAtomMatch`	*builds* match *atoms*
`buildDomAtom`	*builds* dom *atoms*
`addMatchRules`	*builds defining* match *rules*
`addDomRules`	*builds defining* dom *rules*

5.1.2 Using `nfn2dlp`

The interface of `nfn2dlp` is via the command-line. By default, `nfn2dlp` reads the files provided as arguments, treats their contents as one *NFN* program, analyzes its well-formedness and safety, and eventually translates it into a *DLP* program, which will be provided on standard output.

Example 5.1. *Consider the program P represented in the text file* `ex.nfn` *as*

```
a,b(X) :- c(X) ∨ d(X,Y).
c(1).
d(2,3).
```

In order to test for safety and to transform P into a DLP program, we issue

```
$ nfn2dlp.rb ex.nfn
```

on the command line. Since the program is safe, the rewritten program is printed on standard output:

```
a :- auxh1_0(X).
b(X) :- auxh1_0(X).
auxh1_0(X) :- a, b(X).
aux1_0(X) :- c(X).
aux1_0(X) :- d(X,Y).
auxh1_0(X) :- aux1_0(X).
c(1).
d(2,3).
```

The answer sets of the NFN program can be computed by pipelining the output into DLV using the command

```
$ nfn2dlp.rb ex.nfn | DLV --
```

yielding answer set

```
{c(1),d(2,3),a,auxh1_0(1),auxh1_0(2),b(1),b(2),aux1_0(1),aux1_0(2)}.
```

The answer sets of the original NFN program P can be obtained by filtering on the original predicates in P:

```
$ nfn2dlp.rb ex.nfn | DLV -- -filter=a,b,c,d
```

yielding the answer set

```
{c(1),d(2,3),a,b(1),b(2)}
```

As another example we consider an unsafe program.

Example 5.2. *Consider the unsafe program P_u represented in the text file* `ex2.nfn` *as*

```
a,b(X) :- c(X) v not d(X).
```

In order to test for safety and to transform P into a DLP program, we issue

```
$ nfn2dlp.rb ex2.nfn
```

on the command line. Since the program is unsafe, the following message is printed on standard output:

```
Rule a,b(X):-(c(X) v not d(X)). is not safe.
```

5.2 The tool `nfnsolve`

The tool `nfnsolve` computes the answer sets for the *NFN* programs combining systems `nfn2dlp` and DLV.

The architecture of the system `nfnsolve` is shown in Figure 5.12. For `nfnsolve`, all predicate symbols of the *NFN* program are collected during parsing, which are then used to filter the answer sets of the rewritten program computed by the external solver (exploiting the `-filter` option of DLV), which then represent precisely the answer sets of the *NFN* program.

5.2.1 Using `nfnsolve`

Also the tool `nfnsolve` is invoked via command-line interface. As `nfn2dlp`, `nfnsolve` reads the files provided as arguments, and treats their contents as one *NFN* program, analyzes its well-formedness and safety, and eventually translates it into a *DLP* program. In addition, it invokes DLV as a backend. The location of the DLV executable can be specified using the option `-d` or alternatively `--dlv`, the default being `DLV`. Moreover, additional options can be passed to DLV by means of the option `--dlvoptions`; care should be taken that those options should form one word for the shell, which means that usually this option string should be quoted.

Example 5.3. *Continuing Example 5.1 and program P represented in file* `ex.nfn`*, we can issue (provided that the default* `DLV` *is an executable in the path):*

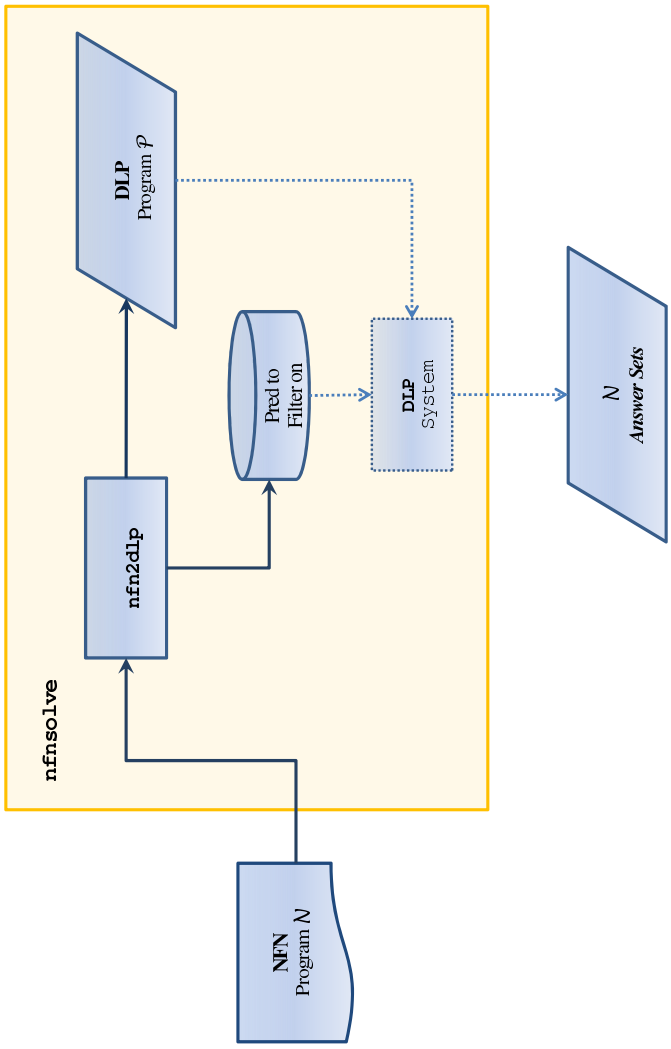

Figure 5.12: System `nfnsolve`

```
$ nfnsolve.rb ex.nfn
DLV [build BEN/Oct 11 2007   gcc 4.1.2]

{c(1),d(2,3),a,b(1),b(2)}
```

If the DLV executable is to be invoked as `./d` *and if this executable is to be passed options* `-silent` *(suppressing the banner with version and compiler information) and* `-nofacts` *(not printing facts), we issue and obtain:*

```
$ nfnsolve.rb -d ./d --dlvoptions '-silent -nofacts' ex.nfn
{a,b(1),b(2)}
```

Chapter 6

Related Work

In this chapter we discuss the relationship to the most important related work.

The chapter is organized as follows:

- ★ In Section 6.1 we show that the answer sets for *NFN* programs coincide with the answer sets defined by Lifschitz, Tang, and Turner in [18] for *NLP* programs, on the common language fragment.
- ★ In Section 6.2 we show that the answer sets for *NFN* programs coincide with the Herbrand stable models defined by Ferraris, Lee, and Lifschitz in [7] for formulas that correspond to *NFN* programs.
- ★ In Section 6.3 we show that the definition of safety for the *NFN* programs is more general than the one defined by Lee, Lifschitz, and Palla in [15] on *NFN* programs, in the sense that there are programs that are unsafe in the definition of [15], but safe in our definition.

6.1 Equivalence to the Semantics of Lifschitz, Tang, and Turner

Lifschitz, Tang, and Turner defined an answer set semantics for (propositional) programs with nested expressions in [18]. We recall their definition adapted for ground *NFN* programs.

Definition 6.1 (Answer Sets of Lifschitz, Tang, and Turner [18])**.** *The reduct* P^{I_L} *of a ground NFN program* P *w.r.t. an interpretation* I *is defined as*

$$\{H^{I_L} \text{ :- } B^{I_L}. \mid H \text{ :- } B. \in P\}$$

where

- *for an atom* a, $a^{I_L} = a$,
- $(\textbf{\textit{not}}\ a)^{I_L} = \begin{cases} \bot, & \text{if } I \not\models (\textbf{\textit{not}}\ a); \\ \top, & \text{if } I \models (\textbf{\textit{not}}\ a); \end{cases}$
- $(l_1, \ldots, l_n)^{I_L} = ({l_1}^{I_L}, \ldots, {l_n}^{I_L})$,
- $(k_1 \vee \cdots \vee k_m)^{I_L} = ({k_1}^{I_L}, \ldots, {k_m}^{I_L})$.

The symbols $\bot$ *and* $\top$ *represent falsity and truth, respectively, where* $I \not\models \bot$ *and* $I \models \top$ *hold for any interpretation* I*. An interpretation* I *is an NLP answer set for a ground NFN program* P *iff it is a minimal model of* P^{I_L}*.*

Example 6.1. *Consider the following NFN program* P*:*

$a.$
$b.$
$f \vee (d, e)$:- $(a \vee \textbf{\textit{not}}\ c).$
p :- $(\textbf{\textit{not}}\ a \vee \textbf{\textit{not}}\ b).$
g :- $(b \vee \textbf{\textit{not}}\ a).$

and interpretation $I = \{a, b, f, g\}$*. Then* P^{I_L} *is the following program:*

$a.$
$b.$
$f \vee (d, e)$:- $(a \vee \top).$
p :- $(\bot \vee \bot).$
g :- $(b \vee \bot).$

We can verify that I *is an NLP answer set of the program* P^{I_L}*.*

In order to prove the equivalence of answer sets, we first show two equivalence results concerning the satisfaction of reducts.

Lemma 6.1.1. *Let r be a ground NFN rule and I a set of atoms, then:*

(1) $I \not\models B(r) \Leftrightarrow I \not\models B(r^{I_L})$;

(2) if r^I *exists,* $\forall J \subseteq I: \quad J \models r^I \Leftrightarrow J \models r^{I_L}$.

Proof. **(1)** The only difference between r and r^{I_L} is on negative literals. But by definition $I \models \ell^{I_L}$ iff $I \models \ell$ for an arbitrary negative literal ℓ, from which the claim follows.

(2) r^I and r^{I_L} differ only in their treatment of body literals. Given a literal ℓ in the body of r, we distinguish (a) $I \models \ell$ and (b) $I \not\models \ell$. In case (a), if ℓ is positive, then ℓ remains both in r^I and r^{I_L}, if $\ell = \textbf{not}\ a$, then $\ell^{I_L} = \top$ while r^I conserves ℓ. However, since $a \notin I$ also $a \notin J \subseteq I$, hence $J \models \ell$. Concerning (b), ℓ is not in r^I. If ℓ is positive and thus remains in r^{I_L}, then $\ell \notin I$ and hence $\ell \notin J \subseteq I$ and $J \not\models \ell$. If ℓ is negative, then it is replaced with $\bot$. In total, we obtain that J satisfies r^I exactly when it satisfies r^{I_L}. □

Theorem 6.1.2. *Given an NFN program P, an interpretation I is an answer set of P according to Definition 2.7 iff I is an NLP answer set of P according to Definition 6.1.*

Proof. ($\Rightarrow$) If I is a minimal model for $Ground(P)^I$, for each $r \in Ground(P)$, s.t. r^I exists, from Observation 1, $I \models B(r)$ and from (2) of Lemma 6.1.1 (for the special case $I = J$), $I \models B(r^{I_L})$ follows. For rules $r \in Ground(P)$, for which no r^I exists, $I \not\models B(r)$ and from (1) of Lemma 6.1.1, $I \not\models B(r^{I_L})$. Consequently I is a model for $Ground(P)^{I_L}$. Moreover, no $J \subset I$ is a model for $Ground(P)^{I_L}$, as it would also be a model for $Ground(P)^I$ because of (2) of Lemma 6.1.1.

($\Leftarrow$) Let I be a minimal model for $Ground(P)^{I_L}$. For each $r^I \in Ground(P)^I$, since $I \models r^{I_L}$ holds by (2) of Lemma 6.1.1, $I \models r^I$ holds as well. As a result, I is a model for $Ground(P)^I$. Furthermore, no $J \subset I$ is a model for $Ground(P)^I$ because J would also be a model for $Ground(P)^{I_L}$. In fact, for each $r^I \in Ground(P)^I$ from (2) of Lemma 6.1.1, $J \models r^{I_L}$. For each $r \in Ground(P)$ s.t. no r^I exists, from Observation 1 $I \not\models B(r)$, therefore a disjunction $D \in B(r)$ exists s.t. $I \not\models D$ iff $I \not\models l$ for all $l \in D$. The corresponding disjunction $D^{I_L} \in B(r^{I_L})$ contains the same atoms of D and

D^{I_L} contains $\bot$ for each negative literal of D. Consequently $J \not\models D^{I_L}$ for all $J \subseteq I$ therefore $J \not\models B(r^{I_L})$ and $J \models r^{I_L}$. □

Since the grounding of a standard program defined in this paper is the same as in [12] we obtain, by virtue of Theorem 6.1.2 and results of [18], the following.

Proposition 6.1.3. *Given a standard DLP program P, the answer sets of P according to Definition 2.7 coincide with the answer sets defined by Gelfond and Lifschitz in [12].*

6.2 Equivalence to Stable Models for First Order Formulas

In this section we show that *NFN* semantics coincides with Herbrand stable models proposed by Ferraris, Lee, and Lifschitz in [7] on *NFN* programs written as first order formulas.

An *NFN* rule H :- B becomes a formula

$$B' \rightarrow H'$$

where B' and H' are obtained from B and H, respectively, by substituting each comma by $\wedge$, and each negative literal of the form **not** a by the formula

$$a \rightarrow \bot$$

An *NFN* program is the first-order formula formed by the conjunction of the universal closures of the transformed rules.

Example 6.2. *Let us consider the NFN program P, consisting of rules:*

$a(X) \vee c(X)$:- $g(X,Y), t(X,Z)$.
b :- $a(X) \vee d(Z)$.

corresponding first-order formula is

$$\forall X,Y,Z((g(X,Y) \wedge t(X,Z)) \rightarrow a(X) \vee c(X)) \wedge \forall X,Z((a(X) \vee d(Z)) \rightarrow b) \quad (6.1)$$

Definition 6.2 (Stable Models of Ferraris, Lee, Lifschitz [7])**.** *Let F be a first-order sentence and let* **p** *be the list of all predicate constants $p_1, \ldots, p_n$ appearing in F. Let $SM[F]$ be the second order formula*

$$F \wedge \neg \exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})) \quad (6.2)$$

where $\mathbf{u}$ *is a list of* n *distinct predicate variables* $u_1, \ldots, u_n$, $\mathbf{u} < \mathbf{p}$ *stands for*

$$\bigwedge_{i=1}^{n} \forall X_1, \ldots X_{\alpha(p_i)} : u_i(X_1, \ldots X_{\alpha(p_i)}) \rightarrow p_i(X_1, \ldots X_{\alpha(p_i)})$$
$$\wedge \neg \bigwedge_{i=1}^{n} \forall X_1, \ldots X_{\alpha(p_i)} : u_i(X_1, \ldots X_{\alpha(p_i)}) \leftrightarrow p_i(X_1, \ldots X_{\alpha(p_i)})$$

where $\alpha(p)$ *denotes the arity of a predicate* p, *and* $F^*(\boldsymbol{u})$ *is defined recursively as*

- $p_i(t_1, \ldots, t_m)^* = u_i(t_1, \ldots, t_m)$;
- $(F \odot G) = (F^* \odot G^*)$, *for* $\odot \in \{\wedge, \vee\}$;
- $(\forall X F)^* = \forall X F^*$;
- $(F \rightarrow G) = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$.

A model of F *is stable if it satisfies* $SM[F]$.

Example 6.3. *Let* $F = a(1) \wedge (\forall X, Z(a(X) \vee d(Z)) \rightarrow b)$ *then*

$$F^* = u_1(1) \wedge \forall X, Z(((u_1(X) \vee u_2(Z)) \rightarrow u_3) \wedge ((a(X) \vee d(Z)) \rightarrow b)) \quad (6.3)$$

and

$$\begin{aligned} SM[F] \equiv\ & a(1) \wedge (\forall X, Z\ (a(X) \vee d(Z)) \rightarrow b) \wedge \\ & \neg(\exists u_1, u_2, u_3 \quad (\forall X_1\ u_1(X_1) \rightarrow a(X_1)) \wedge \\ & (\forall X_1\ u_2(X_1) \rightarrow d(X_1)) \wedge (u_3 \rightarrow b) \wedge \\ & (\exists X_1\ \neg(u_1(X_1) \leftrightarrow a(X_1))) \vee (\exists X_1\ \neg(u_2(X_1) \leftrightarrow d(X_1))) \\ & \vee \neg(u_3 \leftrightarrow b) \wedge u_1(1) \wedge (\forall X, Z\ ((u_1(X) \vee u_2(Z)) \rightarrow u_3) \wedge \\ & ((a(X) \vee d(Z)) \rightarrow b)). \end{aligned}$$

In the following, we will limit ourselves to Herbrand interpretations for formulas, which can be denoted as sets of ground atoms which they satisfy.

Theorem 6.2.1. *Let* P *be an NFN program and* F *its corresponding first-order formula, then* $I \in AS(P)$ *iff* I *is a Herbrand stable model of* F.

Proof. The main observations are that (1) $I \models P$ iff $I \models F$, (2) each structure for $\mathbf{u}$ in $SM[F]$ that satisfies $\mathbf{u} < \mathbf{p}$ corresponds to a $J \subsetneq I$ and vice versa (by substituting predicate names) and (3) $J \models P^I$ iff the corresponding structure for $\mathbf{u}$ it satisfies (together with I) F^*.

In order to see (3), consider first that each formula $\forall \overline{X}\ B \rightarrow H$ for a rule in P is duplicated for F^* to yield $\forall \overline{X}(B^* \rightarrow H^*) \wedge (B \rightarrow H)$. Since

$B \rightarrow H$ contains only original predicates, it is covered by I, and since I must satisfy F for satisfying $SM[F]$, this copy will be satisfied for each possible substitution of rule variables with elements of the Herbrand universe.

Now consider a rule $r\sigma \in \mathit{Ground}(P)$ for which $(r\sigma)^I$ does not exist, that is $I \not\models B(r\sigma)$. That means that there is a $D\sigma \in B(r\sigma)$ such that for each $\ell\sigma$ in $D\sigma$ $I \not\models \ell\sigma$ holds. We show that the corresponding formula in F^* representing D is satisfied for the variable binding corresponding to σ for an interpretation for $\mathbf{u}$ corresponding to an arbitrary $J \subsetneq I$. If ℓ is a positive literal, then $a\sigma \notin I$, hence $a\sigma \notin J \subsetneq I$ and $J \not\models a\sigma$. If $\ell\sigma = \mathbf{not}\ a\sigma$, then F^* contains a formula $(a^* \rightarrow \bot) \wedge (a \rightarrow \bot)$ which is false for the variable binding corresponding to σ because $I \models a\sigma$. In total, if $I \not\models B(r\sigma)$ then the formula in F^* corresponding to $B(r)$ under a variable binding corresponding to σ is not satisfied by I and any interpretation of $\mathbf{u}$ satisfying $\mathbf{u} < \mathbf{p}$.

Next suppose that $(r\sigma)^I$ does exist for a rule $r\sigma \in \mathit{Ground}(P)$. Above we have shown that if $I \not\models \ell\sigma$ for some ground literal $\ell\sigma$ then the formula corresponding to ℓ in F^* is false under a variable binding corresponding to σ w.r.t. I and any interpretation of $\mathbf{u}$. If $I \models \ell\sigma$ for some ground literal $\ell\sigma$, then it will be contained in $(r\sigma)^I$. If ℓ is positive, it occurs unchanged in $r\sigma^I$ and F^*. In case $\ell = \mathbf{not}\ a$, the corresponding formula in F^* will be $(a^* \rightarrow \bot) \wedge (a \rightarrow \bot)$, where the second conjunct is satisfied by I. In this case $J \models \ell\sigma$ iff the interpretation of $\mathbf{u}$ corresponding to J satisfies the formula in F^* corresponding to ℓ under a variable binding corresponding to σ.

Since the structure of conjunctions and disjunctions is preserved in F^*, we can thus conclude (3): Given any I and an interpretation for $\mathbf{u}$ corresponding to $J \subsetneq I$, a rule and a variable binding σ for it, we can see that $J \models (r\sigma)^I$ or $(r\sigma)^I$ does not exist iff I and the interpretation for $\mathbf{u}$ satisfy the formula for r in F^* under a variable binding corresponding to σ. Thus $J \models \mathit{Ground}(P)^I$ iff I and the interpretation for $\mathbf{u}$ satisfy $\exists \mathbf{u}\ (\mathbf{u} < \mathbf{p}) \wedge F^*$. We can thus conclude that I is a minimal model of $\mathit{Ground}(P)^I$ (hence $I \in AS(P)$) iff I satisfies $SM[F]$. □

6.3 Safety for First Order Formulas under Stable Models

Lee, Lifschitz, and Palla in [15] generalize the concept of a safe rule defining safe first-order formulas. Stable models of a safe formula are domain

independent. In this section we show that our safety definition is more general than the one provided in [15] (on the common language fragment).

In Section 6.2 we showed how an *NFN* program can be rewritten in a first-order formula. Next we denote by $\mathcal{F}_{NFN}$ the class of first order formulas corresponding to the class of *NFN* programs.

Definition 6.3 (Safe Formula of Lee, Lifschitz, and Palla [15])**.** *A first-order formula $F = \forall V(G \rightarrow H)$ in $\mathcal{F}_{NFN}$ is safe as of [15] if each variable in F is restricted in G. A variable X is restricted in G according to [15] if X is safe according to an equivalent of Definition 2.9.*

We can thus conclude the following result.

Theorem 6.3.1. *Each first-order formula $F \in \mathcal{F}_{NFN}$ that is safe according to Definition 6.3 is safe according to Definition 2.10. Moreover, there are $F \in \mathcal{F}_{NFN}$ which are safe according Definition 2.10, but not according to Definition 6.3.*

Example 6.4. *Let r be the safe NFN rule a :- $b \vee c(X)$. The corresponding first order formula $\forall X((b \vee c(X)) \rightarrow a$ is unsafe according to Definition 6.3.*

Conclusions

This thesis extends the formalism of Disjunctive Logic Programming under answer set semantics, a very powerful and expressive formalism which is quite popular in the areas of non-monotonic reasoning and logic programming. Although many efficient *DLP* solvers have been developed in the last years, encouraging a number of applications in many real-world contexts, some aspects of *DLP* are limiting. One of them is the simplicity of the language syntax. For this reason some problems have to be represented by programs having a large number of structurally rules. In this way the programs are suboptimal with respect to readability and maintainability.

In this thesis we have introduced *NFN* programs, an extension of non-ground disjunctive logic programs, where conjunctions of atoms and disjunctions of literals are permitted in the heads and bodies of the rules, respectively. We have defined syntax and semantics of the new language and, since ground *NFN* programs are *NLP* programs, we showed that the respective notions of answer sets coincide on this fragment. Moreover, we showed that Herbrand stable models as defined in [7] for first-order formulas that correspond to *NFN* programs coincide with the answer sets for *NFN* programs. Furthermore, we have defined the class of safe *NFN* programs and showed that each program of this class is domain independent, that is, has the same answer sets for each universe containing the constants of the program.

We have developed an algorithm that rewrites a given *NFN* program P into a *DLP* program P_{DLP}. The algorithm runs in polynomial time (thus also the size of P_{DLP} is polynomial in $|P|$) and preserves program safety, and the answer sets of P_{DLP} are in a one-to-one correspondence to the answer sets of P.

We have implemented the system `nfn2dlp` of the presented algorithm using the programming language Ruby and the Treetop framework for parsing. `nfn2dlp` rewrites *NFN* programs into *DLP* programs in the format of DLV,

a state-of-the-art implementation for disjunctive logic programs under the answer set semantics, and thus allows for effective answer set computation of *NFN* programs. Moreover, we implemented the tool `nfnsolve`, which computes the answer sets for the *NFN* programs combining the systems `nfn2dlp` and DLV.

The tools are available at

`http://www.mat.unical.it/software/nfn2dlp/`.

Future work consists of the identification of generalization of the class of *NFN* programs such that the safety definition and algorithm *rewriteNFN* can be reused and extended for that class. Moreover, we examine the possibility of relaxing the safety definition of [15], such that all safe *NFN* programs as defined in this work correspond to safe first-order formulas under the stable model semantics of [7].

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[3] Leopoldo E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 35(2):68–76, 2006.

[4] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[5] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[6] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in AI (LNAI)*, pages 200–212. Springer Verlag, September 2004.

[7] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 372–379, January 2007.

[8] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[9] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 111–122, 2004.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[11] Michael Gelfond and Vladimir Lifschitz. . In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[12] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[13] Tomi Janhunen and Ilkka Niemelä. Gnt - a solver for disjunctive logic programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *Lecture Notes in AI (LNAI)*, pages 331–335. Springer, January 2004.

[14] Balakrishnan V. K. *Schaum's outline of theory and problems of combinatorics*. New York : McGraw-Hill, 1995.

[15] Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 672–676. Springer, 2008.

[16] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[17] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005,*

Proceedings, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[18] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):369–389, 1999.

[19] O. Ore. *Graphs and Their Uses*. New Math Library, 1990.

[20] David Pearce, Vladimir Sarsakov, Torsten Schaub, Hans Tompits, and Stefan Woltran. A Polynomial Translation of Logic Programs with Nested Expressions into Disjunctive Logic Programs: Preliminary Report. In *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, volume 2401 of *Lecture Notes in Computer Science*, pages 405–420. Springer, 2002.

[21] David A. Plaisted and Steven Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[22] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[23] Nathan Sobo. `treetop` homepage. `http://treetop.rubyforge.org/`.

[24] Iain A. Stewart. Complete problems involving boolean labelled structures and projection transactions. *Journal of Logic and Computation*, 1(6):861–882, 1991.

Appendix A

Optimized Translation From *NLP* to *DLP*

The Definition 4.4 holds also if P is a ground *NLP* program. On the contrary, the optimized Definition 4.5 is valid only for ground *NFN* programs. Therefore, in this section, we give a more general optimized definition holding for *NLP* programs.

In the follows, we consider *NLP* program without negative literal in the head.

Definition A.1. *Let P be a ground NLP program without negative literals in the head of the rules, then the corresponding DLP program $\sigma_{opt}(P)$ is defined as follows:*

$$\sigma_{opt}(P) = \{\overline{H(r)} \text{ :- } \overline{B(r)}. \, | r \in P\} \cup \gamma_{opt}(P),$$

where $\overline{H(r)}$ (resp. $\overline{B(r)}$) is $H(r)$ (resp. $B(r)$) if it is a literal otherwise is $\mathbf{L}_{H(r)}$ (resp. $\mathbf{L}_{B(r)}$) and $\gamma_{opt}(P)$ is constructed as follows:

- *for each expression $\phi = (\phi_1, \phi_2)$ occurring in P, γ_{opt} contains the three rules*

$$\mathbf{L}_\phi \text{ :- } \bar{\phi_1}, \bar{\phi_2}. \qquad \bar{\phi_1} \text{ :- } \mathbf{L}_\phi. \qquad \bar{\phi_2} \text{ :- } \mathbf{L}_\phi.$$

- *for each expression $\phi = (\phi_1 \vee \phi_2)$ occurring in the body of some rules of P, γ_{opt} contains the rules*

$$\mathbf{L}_\phi \text{ :- } \bar{\phi_1}. \qquad \mathbf{L}_\phi \text{ :- } \bar{\phi_2}.$$

- *for each expression $\phi = (\phi_1 \vee \phi_2)$ occurring in the head of some rules of P, γ_{opt} contains the rules*

$$\bar{\phi_1} \vee \bar{\phi_2} \text{ :- } \mathbf{L}_\phi. \qquad \mathbf{L}_\phi \text{ :- } \bar{\phi_1}. \qquad \mathbf{L}_\phi \text{ :- } \bar{\phi_2}.$$

where

$$\bar{\phi_i} = \begin{cases} \phi_i, & \text{if } \phi_i \text{ is a literal;} \\ \mathbf{L}_{\phi_i} & \text{otherwise;} \end{cases}$$

$i \in \{1, 2\}$.

Note that if P is a *NLP* program without disjunction in the head then $\sigma_{opt}(P)$ is a normal program and the corresponding computational complexity is smaller than $\sigma(P)$.

Theorem A.0.2. *Let P be an NLP program and $\sigma_{opt}(P)$ be the corresponding DLP program built according to Definition A.1, then*

$$AS(P) = \{I \cap \Pi \,|\, I \in AS(\sigma_{opt}(P))\}.$$

To show the theorem we define the following translation.

Definition A.2. *Let P be an NLP then $\sigma^{\mathbf{L}}_{opt} = \sigma_{opt} \cup \gamma^{\mathbf{L}}_{opt}$ where $\gamma^{\mathbf{L}}_{opt}$ is defined as follows*

- *for each positive literal occurring in P, γ contains the two rules*

$$l \text{ :- } \mathbf{L}_l. \qquad \mathbf{L}_l \text{ :- } l.$$

- *for each negative literal **not** l occurring in P, γ contains the two rules:*

$$l_n \text{ :- } \mathbf{L}_l \qquad \mathbf{L}_l \text{ :- } \textbf{\textit{not}}\ l. \qquad \text{:- } l_n, l.$$

Lemma A.0.3. *Let P be an NLP program with only positive literals in the head rules, $\sigma(P)$ be the standard program built according to Definition 4.4, and $\sigma^{\mathbf{L}}_{opt}(P)$ be the DLP program built according to Definition A.2. Then*

$$AS(\sigma(P)) = AS(\sigma^{\mathbf{L}}_{opt}(P)). \tag{A.1}$$

Proof. From definition of translation σ and $\sigma^{\mathbf{L}}_{opt}$:

$$\sigma(P) \setminus \sigma^{\mathbf{L}}_{opt}(P) = \{\mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_{\phi}. \,|\, \phi_1 \vee \phi_2 \in B(r), \forall\, r \in P\}. \tag{A.2}$$

- $AS(\sigma(P)) \subseteq AS(\sigma^{\mathbf{L}}_{opt}(P))$.
  Suppose that $M \in AS(\sigma(P))$ and $M \notin AS(\sigma^{\mathbf{L}}_{opt}(P))$ then $M \not\models \sigma^{\mathbf{L}}_{opt}(P)$ or M is not a minimal model for $(\sigma^{\mathbf{L}}_{opt}(P))^M$.

  If $M \not\models \sigma^{\mathbf{L}}_{opt}(P)$ then, since $\sigma^{\mathbf{L}}_{opt}(P) \subseteq \sigma(P)$, $M \not\models \sigma(P)$ and this is a contradiction to the hypothesis $M \in AS(\sigma(P))$.

  If M is not a minimal model for $(\sigma^{\mathbf{L}}_{opt}(P))^M$ there exists $N \subsetneq M$ such that $N \models (\sigma^{\mathbf{L}}_{opt}(P))^M$. For each formula $\phi := \phi_1 \vee \phi_2$ occurring in a body rule of P such that $\{\mathbf{L}_{\phi_i} \text{ :- } \mathbf{L}_{\phi}.\} \in (\sigma^{\mathbf{L}}_{opt}(P))^M$, $i = 1, 2$, the formula $\{\mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_{\phi}.\} \in (\sigma(P))^M$. Consequently, if $N \models \mathbf{L}_{\phi_1}$ or $N \models \mathbf{L}_{\phi_2}$ then $N \models \mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_{\phi}$. Moreover, if $N \not\models \mathbf{L}_{\phi}$, $N \models \mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \text{ :- } \mathbf{L}_{\phi}$. As a result, $N \models (\sigma(P))^M$ and this is a contradiction to the hypothesis $M \in AS(\sigma(P))$.

- $AS(\sigma^{\mathbf{L}}_{opt}(P)) \subseteq AS(\sigma(P))$.

  Suppose that $M \in AS(\sigma^{\mathbf{L}}_{opt}(P))$ and $M \notin AS(\sigma(P))$. If $M \not\models \sigma(P)$ then there exists a rule $r \in (A.2)$ such that $M \models \mathbf{L}_{\phi}$ and $M \not\models \mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2}$. That is, $M \not\models \mathbf{L}_{\phi_1}$ and $M \not\models \mathbf{L}_{\phi_2}$. Therefore, $M \not\models \mathbf{L}_{\phi_i} \text{ :- } \mathbf{L}_{\phi}$, $i = \{1, 2\}$, where $\mathbf{L}_{\phi_i} \text{ :- } \mathbf{L}_{\phi} \in \sigma^{\mathbf{L}}_{opt}(P)$, contradicting hypothesis $M \in AS(\sigma^{\mathbf{L}}_{opt}(P))$.

  If M is not a minimal model for $(\sigma(P)^M)$, there exists $N \subsetneq M$ such that $N \models (\sigma(P)^M)$. Since $\sigma^{\mathbf{L}}_{opt}(P) \subseteq \sigma(P)$ then $\sigma^{\mathbf{L}}_{opt}(P)^M \subseteq \sigma(P)^M$. As a result, $N \models (\sigma^{\mathbf{L}}_{opt}(P)^M)$ and this is a contradiction to the hypothesis $M \in AS(\sigma^{\mathbf{L}}_{opt}(P))$.

□

Proof. (Theorem A.0.2) From Lemma A.0.3, since $AS(\sigma^{\mathbf{L}}_{opt}(P)) = AS(\sigma(P))$, then

$$AS(P) = \{I \cap \Pi \,|\, I \in AS(\sigma^{\mathbf{L}}_{opt}(P))\}.$$

Moreover, it easy to see that,

$$AS(\sigma_{opt}(P)) = \{I \cap \mathbf{L} \;\; | \, I \in AS(\sigma^{\mathbf{L}}_{opt}(P))\}$$

where $\mathbf{L} = \{\mathbf{L}_{\phi} \mid \phi \text{ is a literal of } P\}$, and the theorem holds. □

Appendix B

Classes for Handling Rewriting

In this section we report the full implementation for the classes REWRITEHEAD and REWRITEBODY described in Section 5.1.1.

class RewriteHead

```
require 'disjunction_class'
require 'rule'
class RewriterHead
  attr :nfn_head
  attr :dlp_head
  attr :std_rules_chr

  def initialize(head)
    @nfn_head = head
    @dlp_head = Disjunction.new([ ])
    @std_rules_chr = [ ]
  end

  def rewrite_head(number_rule)
    i = 1
    @nfn_head.each do |bc|
      if bc.size == 1 then
        @dlp_head.add(bc.to_atom)
      else
        terms = Terms.new([])
        bc.each do |a|
```

```
        terms = terms + a.getTerms
      end
      terms = terms.deleteDuplicates
      pred = auxh + i.to_s + '_' + number_rule.to_s
      aux = Atom.new(pred,terms)
      # builds auxilary rules to define
      # the auxiliry predicate
      bc.each do |a|
        aux_r = Rule.new(a,aux)
        @std_rules_chr = @std_rules_chr + [aux_r.to_s]
      end
      @std_rules_chr = @std_rules_chr + [Rule.new(aux,bc).to_s]
      @dlp_head.add([aux])
      i+=1
      end
    end
  end
 end
```

class RewriteBody

```
require 'rule'
$UNRES = Term.new(unRes")"
class RewriterBody
  attr :nfn_body #formula in CNF
  attr :dlp_body #conjunction of literals
  attr :std_rules_chr #array with standard rule
  attr :Univ #array of predicate Univ
  attr :aux_hash #basic disjunction with corresponding atom
  attr :match #array match atoms
  attr :UnResSharedVars

  def initialize(b)
    @nfn_body = b
    @dlp_body = Conjunction.new([])
    @std_rules_chr = Array.new
    @Univ = Array.new
```

```
  @aux_hash = Hash.new
  @match = Array.new
  @UnResSharedVars = Terms.new([])
end

# Given a basic disjunction, function build_aux builds
# a new atom auxiliary  with fresh predicate name,
# representing the basic disjunction.
# @bd: basic disjunction
# @n_rule: number of the NFN rule containing bd
def build_aux(bd, n_rule, head)
  n_disj = bd.num_bd
  pred = 'aux' + n_disj.to_s + '_' + n_rule.to_s
  terms = Terms.new([])
  safe_shared = (bd.safe_vars & @nfn_body.shared_vars) +
                (bd.safe_vars & head.vars_disj)
  safe_shared = safe_shared.deleteDuplicates
  terms = safe_shared + bd.sharedVars + bd.unResVars
  return Atom.new(pred,terms)
end

# Function atomsJoiningUnresVars
# 1. renaming unrestricted variables appearing in each
#     auxiliary atom;
# 2. build fresh atoms match" to define an explicit match"
#    between unrestricted variables and contant $UNRES;
# 3. adds rules defining the predicate match.
# @nr: number rule
def atomsJoiningUnresVars(nr)
  @nfn_body.unRes_vars.each do |sv|
    pred = 'match' + sv.term.to_s + '_' + nr.to_s
    svArray = Array.new
    @aux_hash.each_key do |bd1|
      if bd1.unResVars.include?(sv)
         # rename sv appearing in the
         # corresponging auxiliary atom of bd1
         svNew = sv.term.to_s + bd1.num_bd.to_s + '_' + nr.to_s
```

```
        svNewT = Term.new(svNew)
        aux_hash[bd1].terms.set_term(aux_hash[bd1].
                              terms.index(sv),svNewT)
        svArray.push(svNew)
      end
    end
    tmp=svArray.length
    while tmp > 0
      (tmp-1).times{|i|
      if svArray[i] > svArray[i+1]
        svArray[i],svArray[i+1] = svArray[i+1],svArray[i]
       end
       }
      tmp-=1
  end
  svArray
  # build atom match to define macching between X and  $UNRES
  varCorr = svArray[0]
  for n in 1.. svArray.size - 1
    termsMatch = Terms.new([])
    termsMatch.set_term(0, Term.new(varCorr))
    termsMatch.set_term(1, Term.new(svArray[n]))
    varCorr = sv.term.to_s + (n-1).to_s + n.to_s + '_' + nr.to_s
    termsMatch.set_term(2, Term.new(varCorr))
    @match = @match + [Atom.new(pred,termsMatch)]
  end
  bodyMatch = Atom.new('univ' + sv.term.to_s  + '_' +
             nr.to_s,[sv])
  addUnivUnresRules(bodyMatch)
  # Add rule defining match predicate
  @std_rules_chr = @std_rules_chr +
  [Rule.new(Atom.new(pred,
                 Terms.new([sv,sv,sv])),bodyMatch).to_s]
  @std_rules_chr = @std_rules_chr +
  [Rule.new(Atom.new(pred,
                 Terms.new([sv,$UNRES,sv])),bodyMatch).to_s]
  @std_rules_chr = @std_rules_chr +
```

```
  [Rule.new(Atom.new(pred,Terms.new([$UNRES,sv,sv])),
                bodyMatch).to_s]
  @std_rules_chr = @std_rules_chr +
                [Rule.new(Atom.new(pred,Terms.new(
                [$UNRES,$UNRES,$UNRES])),[]).to_s]
   end
end


# Function addUnivUnresRules build rules defining
# univ predicate of unrestricted variables.
#
# @univ: predicate to define
def addUnivUnresRules(univ)
  nfn_body.each do |bd|
    bd.each do |lit|
      if  lit.isNegative == false &&
       lit.internal_atom.terms.include?(univ.terms[0])
       @std_rules_chr = @std_rules_chr+
                          [Rule.new(univ, lit).to_s]
      end
    end
  end
end


# Function addUnivSharedRule build rules defining
# univ predicate of safe variables.
#
def addUnivSharedRules
  @Univ.each do |univ|
    body = Conjunction.new([])
    # atom univ contains only a variable
    var = univ.terms[0]
    # an aux atom is added to body only if var
    # is a safe variable of the corrisponding
    # basic disjunction
```

```
    @aux_hash.each_key  do |bd|
    num = bd.num_bd
    if bd.safe_vars.include?(var)
      body = Conjunction.new([aux_hash[bd]])
      @aux_hash.each_key  do |bd2|
        if bd2.safe_vars.include?(var)
          if bd2.num_bd < num
            body = Conjunction.new([aux_hash[bd2]])
          end
          num = bd2.num_bd
        end
      end
    break
    end
   end
   @std_rules_chr = @std_rules_chr +
                  [Rule.new(univ,body).to_s]
  end
end


# Function atomsJoiningSharedVars
#
# 1. it renames each shared variables X assuming
#    value $UnRes in all auxiliary atoms
# 2. build fresh atoms 'match' to define an explicit match
#    between shared variables and contant $UNRES;
# 3. adds rules defining the predicate match.
#
# @nr: number rule
def atomsJoiningSharedVars(nr)
  @aux_hash.each_key do |bd|
    nd = bd.num_bd
    bd.sharedVars.each do |sv|
    # rename each variable shared assuming value $UNRES
    # appearing in the corresponging auxiliary atom of bd
    if @UnResSharedVars.include?(sv)
```

```
        svNew = sv.to_s + nd.to_s + '_' + nr.to_s
        svNewT = Term.new(svNew)
        @aux_hash[bd].terms.set_term(aux_hash[bd].terms.
                                     index(sv),svNewT)
        pred = 'match' + sv.to_s  + '_' + nr.to_s
        match_atom = Atom.new(pred, Terms.new([sv,svNew]))
        @match = @match + [match_atom]
        univ_atom = Atom.new('univ' + sv.term.to_s + '_' +
                                          nr.to_s,[sv])
        # if univ_atom does not contained in @Univ
        # then it is added to @Univ
        add = true
        @Univ.each do |a|
          if (a == univ_atom)
            add = false
          end
        end
        if add == true
          @Univ = @Univ + [univ_atom]
        end
        @std_rules_chr = @std_rules_chr +
         Rule.new(Atom.new(pred,Terms.new([sv,sv])),
                                     univ_atom).to_s]
        @std_rules_chr = @std_rules_chr +
        [Rule.new(Atom.new(pred,Terms.new([sv,$UNRES])),
                                      univ_atom).to_s]
     end
    end
  end
end


 # Given an auxiliary atom, function build_aux_rules
 # builds the rules that define the auxiliary predicate
 # The function substitutes constant $UNRES for each
 # variable appearing only in the head of the rule.
 #
```

```
# @a: auxiliary atom
# @bd: basic disjunction
# @nd: number disjunction
def build_aux_rules(a,bd,nd,nr)
  # for each literal of bd a new rule is built
  bd.each do |l|
    head_aux_rule = a.copy
    body_aux_rule = Conjunction.new([l])
    # unbound is the array of variables appearing in
    # head_aux_rule and that do not appear in the
    # literal l
    unbound = head_aux_rule.terms - l.internal_atom.terms
    # each variable of unbound in aux is substituted
    # with the constant #unRes to build a safe rule
    unbound.each do |t|
      head_aux_rule.terms.set_term(head_aux_rule.
                                        terms.index(t),$UNRES)
      # Each safe shared variable assuming value $UNRES
      # is added to @UnResSharedVars
      @UnResSharedVars = @UnResSharedVars.push(t)
    end
    # il l is a negative literal
    # for each shared variable of l a new atom
    # is added to body_aux_rule in order to build
    # a safe rule
    if l.isNegative == true
      l.internal_atom.terms.each do |sv|
      pred = 'univ' + sv.to_s + '_' + nr.to_s
      ts = Terms.new([sv])
      univ_atom = Atom.new(pred,ts)
      body_aux_rule.add(univ_atom)
      # if univ_atom does not contained in @Univ
      # then it is added to @Univ
      add = true
      @Univ.each do |a|
        if a == univ_atom
          add = false
```

```
        end
      end
      if add == true
        @Univ = @Univ + [univ_atom]
      end
   end
  end
  @std_rules_chr = @std_rules_chr + [(Rule.new(head_aux_rule,
                        body_aux_rule)).to_s]
 end
end


 # Function rewrite_body builds corresponding standard body
 # @dlp_body of NFN body rule @dlp_body
 def rewrite_body(number_rule,head)
  num_disj = 1
  @nfn_body.each do |bdis|
     bdis.set_num_bd(num_disj)
     if bdis.size == 1 then
   # since the  basic disjunction contains only a literal
   # the aux atom is not built
   @dlp_body.add(bdis.to_literal)
   if !bdis[0].isNegative
     # the atom is added to aux_hash because all its variables
     # are safe
     aux = Atom.new(bdis[0].internal_atom.predicate_name,
            bdis[0].internal_atom.terms)
     @aux_hash = aux_hash.merge({bdis => aux})
   end
   else
     # auxiliary atom corresponding to the basic disjunction
     # is built
     aux = build_aux(bdis,number_rule, head)
     @aux_hash = @aux_hash.merge({bdis => aux})
     build_aux_rules(aux, bdis, num_disj, number_rule)
   end
```

```
    num_disj += 1
   end    # the function builds match atoms for shared variables
   # and rename shared variables in each auxiliary atom
   atomsJoiningSharedVars(number_rule)
   addUnivSharedRules
   # the function builds match atoms for unrestricted variables
   # and rename unrestricted variables in each auxiliary atom
   atomsJoiningUnresVars(number_rule)
   @aux_hash.each_key do |bd|
      if bd.size > 1
        @dlp_body.add(aux_hash[bd])
      end
    end
    @match.each do |lit|
      @dlp_body.add(lit)
    end
  end

end
```