nfn2dlp and nfnsolve: Normal Form Nested Programs Compiler and Solver

Annamaria Bria, Wolfgang Faber, and Nicola Leone

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{`a.bria,faber,leone`}`@mat.unical.it`

Abstract. Normal Form Nested (*NFN*) programs have recently been introduced in order to allow for enriching the syntax of disjunctive logic programs under the answer sets semantics. In particular, heads of rules can be disjunctions of conjunctions, while bodies can be conjunctions of disjunctions. Different to many other proposals of this kind, *NFN* programs may contain variables, and a notion of safety has been defined for guaranteeing domain independence. Moreover, *NFN* programs can be efficiently translated to standard disjunctive logic programs (*DLP*).

In this paper we present the tool `nfn2dlp`, a compiler for *NFN* programs, which implements an efficient translation from safe *NFN* programs to safe *DLP* programs. The answer sets of the original *NFN* program can be obtained from the answer sets of the transformed program (which in turn can be obtained by using a *DLP* system) by a simple transformation. The system has been implemented using the object-oriented programming language Ruby and Treetop, a language for Parsing Expression Grammars (PEGs). It currently produces *DLP* programs in the format of DLV. The separate script `nfnsolve` uses DLV as a back-end to compute answer sets for *NFN* programs. Thus, combining the two tools we obtain a system which supports the powerful *NFN* language, and is available for experiments.

1 Introduction

Disjunctive logic programming under the answer set semantics (*DLP*, *ASP*) has been acknowledged as a versatile formalism for knowledge representation and reasoning during the last decade. The heads (resp. the bodies) of *DLP* rules are disjunctions (resp. conjunctions) of simple constructs, viz. atoms and literals. In [1], we proposed Normal Form Nested programs that are an extension of Disjunctive Logic Programs with variables. In particular the head of an *NFN* rule is a formula in disjunctive normal form; while the body is a formula in conjunctive normal form. We provided also a polynomial translation from *NFN* programs to *DLP* programs. The main idea of the algorithm is to introduce new atoms in order to rewrite conjunctions appearing in the head of the rules and disjunctions appearing in the bodies. This result allows for evaluating *NFN* programs using DLP systems, such as DLV [2], GnT [3], or Cmodels3 [4].

In this paper we describe a tool implementing the efficient translation from safe *NFN* programs to safe *DLP* programs presented in [1], called `nfn2dlp`. The system provides an *NFN* parser and safety checker, and an efficient translation to an equivalent *DLP* program. 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. A second tool, called `nfnsolve`, automates this procedure and directly computes answer sets for *NFN* programs by translating them into *DLP* programs (in the same way as `nfn2dlp`), and then invoking DLV on them, filtering out all symbols that have been introduced during the translation to produce the answer sets of the input *NFN* programs.

2 Normal Form Nested Programs

In this section, we briefly introduce syntax, semantics and safety of *NFN* programs. For a more detailed discussion, we refer to [1].

Syntax We consider a first-order language without function symbols. *NFN* programs are finite sets of rules of the form

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

where each of $C_1, \ldots, C_n$ is a positive basic conjunction $(a_1, \ldots, a_n)$ of atoms $a_1, \ldots, a_n$ and each of $D_1, \ldots, D_m$ is a basic disjunction $(l_1 \vee \ldots \vee l_n)$ of literals $l_1, \ldots, l_n$. The parentheses around basic conjunctions and disjunctions may be omitted. $C_1 \vee \ldots \vee C_n$ is the *head*, and $D_1, \ldots, D_m$ is the *body* of a rule. An *NFN* program is called *standard* if all basic conjunctions and disjunctions are singleton literals.

In our experience, the need for going beyond DLP arises relatively often in real world applications. As an example, we recall a consistent query answering setting from [1]: According to [5], a global relation $p(ID, name, surname, age)$ (for persons) with a key-constraint on the first attribute ID is "repaired" by intensionally deleting one of them whenever two tuples would share the same key. In DLP, this is done by the following rules ($\overline{p}$ denotes deleted tuples, p' the resulting consistent relation).

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

The first three DLP rules can be written as a single *NFN* rule.

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

Safety Let r be an *NFN* rule. A variable X in r is *restricted* if there exists a positive basic disjunction D in the body of r, such that, for each $a \in D$, X occurs in a; we also say that D saves X and X is made safe by D. A rule is safe if each variable appearing in the head and each variable that appears in a negative body literal are restricted. An *NFN* program is safe if each of its rules is safe.

Safe programs have the important property of domain independence, that is, their semantics is invariant with respect to the given universe (as long as it is large enough).

Semantics We consider ground instantiations of NFN programs with respect to a given universe. When considering safe NFN programs, the Herbrand universe is sufficient. An *interpretation* for a safe NFN program P can therefore be denoted as a subset of the Herbrand base. The satisfaction of ground rules by interpretations is defined in the classical way, interpreting rules as implications. An interpretation that satisfies a program is called a *model*.

The *reduct* of a ground program P with respect to an interpretation I, denoted by P^I, is obtained by (1) deleting all false literals w.r.t. I from rule bodies, and (2) deleting all rules s.t. any basic disjunction becomes empty after (1). An interpretation I is an *answer set* for P iff I is a subset-minimal model for P^I. We denote the set of answer sets for P by $AS(P)$.

3 An Efficient Translation from *NFN* to *DLP*

In this section we will review the rewriting algorithm *rewriteNFN* from [1], to which we refer for a more detailed description. The basic structure of *rewriteNFN* is shown in Fig. 1. The input for *rewriteNFN* is a safe NFN program P and it builds and eventually returns a safe standard DLP program, P_{DLP}. The algorithm transforms one rule at a time. For each NFN rule, it constructs one *major rule*, which maintains the structure of the NFN rule, replacing complex head and body structures by appropriate labels. Head and body of the major rule are built independently by means of functions *buildHead* and *buildBody*, respectively, which will be described in the sequel of this section. These functions may also create a number of auxiliary rules, for defining labels and auxiliary predicates which are needed mostly for guaranteeing safety of the transformed program.

```
begin rewriteNFN
Input: $NFN$ program $P$
Output: $DLP$ program $P_{DLP}$.
var $B$: conjunction of literals; $H$: disjunction of atoms;
  $P_{DLP} := \emptyset$;
  for each rule $r \in P$ do
    $H :=$ buildHead$(H(r), P_{DLP})$;
    $B :=$ buildBody$(B(r), P_{DLP})$;
    $P_{DLP} := P_{DLP} \cup \{H$ :- $B.\}$;
  return $P_{DLP}$;
```

Fig. 1. Algorithm *rewriteNFN*

3.1 Head Transformation

Function *buildHead* is comparatively lightweight and replaces non-singular nested structures by fresh label atoms. For each head conjunction C of a rule r containing more than one atom, a label atom with the fresh predicate name $auxh^r_C$ and all variables in C is created in its place. In order to act as a substitute for C, the function also creates auxiliary rules $auxh^r_C(\ldots)$:- $C.$ and a_i :- $auxh^r_C(\ldots).$ for each $a_i \in C$. The safety of the auxiliary rules is straightforward, and the safety of the major rule is guaranteed by the safety of the original NFN program and the body transformation described next.

3.2 Body Transformation

More care has to be taken in function *buildBody*. Since not all variables in a safe NFN rule body have to be restricted, just replacing body disjunctions by labels as for NFN heads may result in an unsafe auxiliary rule because of an unrestricted variable. If the variable in question occurs only in its body disjunction, it can be safely dropped from the label atom, but if this variable occurs also elsewhere in the rule, the values it represents must match in each of its occurrences, while in some occurrences the variable may not be bound to any value. Therefore, *buildBody* focuses on *shared variables*, where a variable X is *shared* in a rule r, if it appears in two different body disjunctions of r, or if X appears in both head and body of the rule.

For creating the body of the major rule, *buildBody* replaces each body disjunction D of a rule r containing more than one literal by a label atom $\mathit{aux}^r_D(V_1, \ldots, V_n)$, where aux^r_D is a fresh symbol and $V_1, \ldots, V_n$ are the shared variables of r occurring in D. An auxiliary rule for defining $\mathit{aux}^r_D(V_1, \ldots, V_n)$ is added for each literal in D, where variables not occurring in a literal are replaced by the special constant $\#u$, representing that the respective variable is not bound in this occurrence. Moreover, if the literal is negative, some new *universe* atoms (see [1]) are added to the body defining the label atom, which in turn are defined by appropriate auxiliary rules. Since $\#u$ has to match with any other constant, matching has to be made explicit in the body of the major rule by adding dedicated atoms, which are also defined by auxiliary rules.

3.3 Properties of the Algorithm

Let P a safe NFN program, $P_{DLP} = \mathit{rewriteNFN}(P)$, and $\mathcal{A}_N$ and $\mathcal{A}_D$ be the sets of predicate symbols that appear in P and in P_{DLP}, respectively ($\mathcal{A}_N \subseteq \mathcal{A}_D$). Then, there is a bijection between $AS(P_{DLP})$ and $AS(P)$ such that $J \in AS(P_{DLP})$ iff $J \cap \mathcal{A}_N \in AS(P)$. As mentioned previously, all rules generated by *rewriteNFN* are safe. Moreover, the complexity of the algorithm is a small polynomial.

4 Systems `nfn2dlp` and `nfnsolve`

Algorithm *rewriteNFN*, along with an NFN parser and safety checker has been implemented as a front-end to *DLP* systems. Currently, the syntax of the system DLV is supported, but the implementation is decoupled from DLV and can easily be modified for supporting other *DLP* systems such as GnT or Cmodels3. The resulting tools, called `nfn2dlp` (for translating only) and `nfnsolve` (for additionally invoking a *DLP* backend), are publicly available at `http://www.mat.unical.it/software/nfn2dlp/` . In the following we provide some information about issues in the implementation mainly of `nfn2dlp`. Moreover, we give a description of the usage of `nfn2dlp` and `nfnsolve`.

4.1 Implementation of `nfn2dlp` and `nfnsolve`

The tools `nfn2dlp` and `nfnsolve` have been implemented using the language Ruby [6], an object-oriented language rooted also in functional and scripting languages.

Both tools exploit an *NFN* parser, implemented using the tool `treetop` [7], which provides a parser generator for Parsing Expression Grammars (PEGs) [8] for Ruby. PEGs are a novel concept for parser specification, which look similar to classical grammars but differ in semantics; most importantly these grammars avoid ambiguity.

The tools rely on a code base which has been constructed using an object-oriented design: For all language constructs, such as atoms, literals, basic disjunctions, basic conjunctions and rules, appropriate Ruby classes exist, and the respective objects are created during parsing. The safety check has been implemented as a method of the rule class.

Moreover, two classes for handling rewriting have been defined, RewriteHead and RewriteBody, respectively. These classes contain as attributes the respective *NFN* structure (head and body, respectively), a corresponding *DLP* structure for constructing the major rule, and a set of auxiliary *DLP* rules. The methods of these classes effectively implement *buildHead* and *buildBody*.

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.

Both `nfn2dlp` and `nfnsolve` provide a basic commandline interface, which we overview in Sections 4.2 and 4.3.

4.2 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 1. Consider the program P represented in the text file `ex.nfn` as

$$a, b(X) \text{ :- } c(X) \vee d(X, Y). \quad c(1). \quad d(2, 3).$$

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:

$$\begin{array}{llll} a \text{ :- } auxh1\_0(X). & b(X) \text{ :- } auxh1\_0(X). & auxh1\_0(X) \text{ :- } a, b(X). & c(1). \\ aux1\_0(X) \text{ :- } c(X). & aux1\_0(X) \text{ :- } d(X, Y). & auxh1\_0(X) \text{ :- } aux1\_0(X). & d(2, 3). \end{array}$$

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 $\{c(1), d(2,3), a, b(1), b(2)\}$.

4.3 Using `nfnsolve`

Also `nfnsolve` possesses a 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 following option `-d` or alternatively `--dlv`, the default being `DLV` in the path. Moreover, additional options can be passed on 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 those options should be quoted.

Example 2. Continuing Example 1 and program P represented in file `ex.nfn`, we can issue (provided that the default `DLV` is an executable in the path):

```
$ 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)}
```

References

1. Bria, A., Faber, W., Leone, N.: Normal form nested programs. In: Proceedings of the 11th European Conference on Logics in Artificial Intelligence (JELIA 2008). LNCS 5293, (2008) 76–88
2. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
3. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, (2004) 331–335
4. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
5. Bertossi, L.E.: Consistent query answering in databases. SIGMOD Record **35**(2) (2006) 68–76
6. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly (2008)
7. Sobo, N.: `treetop` homepage `http://treetop.rubyforge.org/`.
8. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL 2004. (2004) 111–122