nfn2dlp: A Normal Form Nested Programs Compiler*

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, previous results show that there is a polynomial translation from NFN programs 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 and can use DLV as a back-end.

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, which represent conjunctions appearing in the head of the rules and disjunctions appearing in the bodies. This translation 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

* Supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.”

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.

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 $(a_1, \ldots, a_k)$ of atoms and each of $D_1, \ldots, D_m$ is a basic disjunction $(l_1 \vee \ldots \vee l_j)$ of literals. 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.

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.

As shown in [1], safe programs have the important property of domain independence, that is, their semantics is invariant with respect to the considered universe.

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]. 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* takes as input an NFN rule head and builds a corresponding standard rule head (disjunction of atoms). For each non-singular basic conjunction $C = a_1, \ldots, a_n$ (i.e. $n > 1$), the function builds an auxiliary atom representing C. The predicate name of the new atom is $auxh^r_C$ and its arguments are all variables occurring in C. In order to act as a substitute for C, the function also creates auxiliary rules $auxh^r_C(\ldots)$:- $a_1, \ldots, a_n$. and a_i :- $auxh^r_C(\ldots)$., $i = 1, \ldots, n$. It is easy to see that these auxiliary rules are safe.

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 unsafe auxiliary rules because of unrestricted variables. 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. 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. As a further complication, in some literals a shared variable may not be bound to any value.

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 $aux^r_D(V_1, \ldots, V_n)$, 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 $aux^r_D(V_1, \ldots, V_n)$ is added for each literal in D, where variables not occurring in the respective literal are replaced by the special constant $\#u$, representing that the respective variable is unbound. Moreover, if the literal is negative, some new *universe* atoms are added to the body 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 *matching* atoms, 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, $I \in AS(P)$ if and only if there exists a unique $J \in AS(P_{DLP})$ such that $I = J \cap \mathcal{A}_N$. As mentioned previously, all rules generated by *rewriteNFN* are safe. Moreover, the complexity of the algorithm is a small polynomial.

4 The `nfn2dlp` System

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 tool, called `nfn2dlp`, is publicly available at

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

In the following we provide some information about issues in the implementation of `nfn2dlp`. Moreover, we give a description of the usage of `nfn2dlp`.

4.1 Implementation of `nfn2dlp`

The tool `nfn2dlp` has been implemented using the language Ruby [5], an objected-oriented language rooted also in functional and scripting languages.

The tool comprises an NFN parser, implemented using the tool `treetop` [6], which provides a parser generator for Parsing Expression Grammars (PEGs) [7] for Ruby. PEGs are a novel concept for parser specification, which look similar to BNF grammars but differ in semantics; most importantly these grammars avoid ambiguity and allow for modular language specification. These properties simplify the development and allow for reuse of the parser.

The tool `nfn2dlp` 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 have been designed. 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*. Finally, a basic command-line interface is provided, which we overview in Section 4.2.

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 `example.txt` 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 example.txt
```

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

$$a \text{ :- } auxh1\_0(X). \quad b(X) \text{ :- } auxh1\_0(X). \quad auxh1\_0(X) \text{ :- } a, b(X). \quad c(1).$$
$$aux1\_0(X) \text{ :- } c(X). \quad aux1\_0(X) \text{ :- } d(X, Y). \quad auxh1\_0(X) \text{ :- } aux1\_0(X). \quad d(2, 3).$$

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

```
$ nfn2dlp.rb example.txt | DLV -- -filter=a,b,c,d
```

yielding the answer set $\{c(1), d(2, 3), a, b(1), b(2)\}$. The distribution also provides a script `nfnsolve` which automates the call to DLV with the appropriate filter.

5 Future Work

We plan to modify `nfn2dlp` in order to output the resulting DLP program in formats acceptable to other systems (`lparse` and `gringo` acting as grounders for Cmodels3, GnT, and ClaspD). Furthermore, we are working on extensions of the NFN language in order to allow for a richer (not necessarily normal-form) syntax in heads and bodies. This will require an adaptation of the safety definition and an extension of the translation algorithm.

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. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly (2008)
6. Sobo, N.: `treetop` homepage `http://treetop.rubyforge.org/`.
7. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004). (2004) 111–122