Handling Function Symbols in the DLV Grounder

Francesco Calimeri[1], Susanna Cozza[2], and Simona Perri[1]

[1] Department of Mathematics, University of Calabria,
P.te P. Bucci, Cubo 30B, I-87036 Rende, Italy
[2] DEIS, University of Calabria,
P.te P. Bucci, Cubo 42C, I-87036 Rende, Italy
{calimeri,cozza,perri}@mat.unical.it

Abstract. This paper describes how the grounder of the ASP system DLV handles programs with function symbols, briefly illustrating two different strategies alongside the results of a preliminary experimental activity. The grounder is able to deal with ASP programs making an unrestricted use of function symbols, and it can perform an on-demand check in order to ensure that the input falls into the class of *argument-restricted* programs, thus guaranteeing the termination.

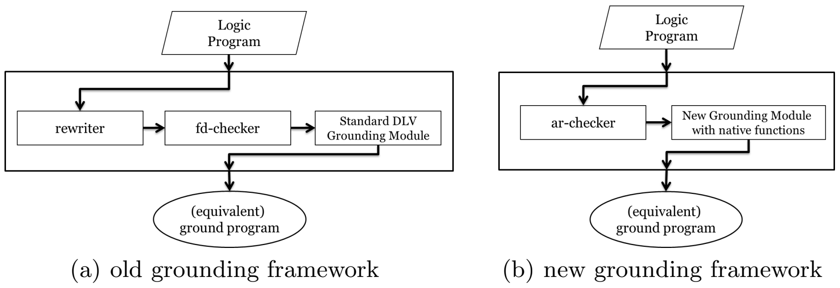
1 Introduction

Answer Set Programming (ASP) is a highly expressive language, which, due to the presence of disjunction and negation, allows the use of nondeterministic definitions for modelling complex problems in computer science, in particular in Artificial Intelligence. Traditionally, ASP has often been used as a first-order language without function symbols, similar to Datalog, in order to deal with finite structures only. More recently, also uninterpreted function symbols have been frequently considered, thus enabling a natural representation of complex and recursive structures, such as strings, lists, stacks, trees and many other. In the latest years, ASP became widely used for knowledge representation and reasoning in many application scenarios, also thanks to the blooming of robust and efficient systems. Some of these systems started to support function symbols to some extent. Unfortunately, the presence of function symbols within ASP programs has a strong impact on the grounding process, which might even not terminate: the common reasoning tasks are indeed undecidable, in the general case.

In this paper we discuss some advances recently introduced in the grounder of the ASP system DLV [9], concerning the management of programs with functions. In particular, a new approach, relying on functional terms as native structures, has been implemented, which brings relevant performance improvements w.r.t. the previous one, which was based on rewriting. Moreover, a new checker has been implemented, that allows the user to statically recognize if the input program belongs to a class for which the termination of the grounding process can be guaranteed: the class of *argument-restricted* programs [10], which is larger than the class recognized by DLV so far. The results of an experimental analysis are also reported, confirming a significant gain in terms of efficiency w.r.t. the old approach.

2 Preliminaries

This Section shortly reports some preliminaries about the language herein considered, i.e. ASP with functions.


Logic Program
rewriter
fd-checker
Standard DLV Grounding Module
(equivalent) ground program
Logic Program
ar-checker
New Grounding Module with native functions
(equivalent) ground program

(a) old grounding framework (b) new grounding framework

Fig. 1. New and old System Architectures

A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \dots t_n$ are terms and f is a function symbol (*functor*) of arity n, then $f(t_1, \dots, t_n)$ is a *functional term*.

If $t_1, \dots, t_k$ are terms, then $p(t_1, \dots, t_k)$ is an *atom*, where p is a predicate of fixed arity $k \geq 0$; by $p[i]$ we denote its i-th argument. Atoms prefixed by # are instances of *built-in* predicates: such kind of atoms are evaluated as true or false by means of operations performed on their arguments, according to some predefined semantics.[3]

A *literal* l is of the form a or `not` a, where a is an atom; in the former case l is *positive*, and in the latter case *negative*. A *rule* r is of the form $\alpha_1 \vee \cdots \vee \alpha_k$:- $\beta_1, \dots, \beta_n$, `not` $\beta_{n+1}, \dots,$ `not` β_m. ($m \geq 0$, $k \geq 0$; $\alpha_1, \dots, \alpha_k$ and $\beta_1, \dots, \beta_m$ being atoms). We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ (the *head* of r) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of r), where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ (the *positive body* of r) and $B^-(r) = \{$`not` $\beta_{n+1}, \dots,$ `not` $\beta_m\}$ (the *negative body* of r). If $H(r) = \emptyset$ then r is a *constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then r is referred to as a *fact*.

A rule is safe if each variable in it appears in at least one positive literal in the body. For instance, the rule $p(X, f(Y, Z))$:- $q(Y),$`not` $s(X)$. is not safe, because of both X and Z. From now on we assume to deal only with safe rules. An ASP program is a finite set P of rules. As usual, a program (a rule, a literal) is said to be *ground* if it contains no variables.

A thorough discussion about the semantics, based on the notion of Answer Set proposed in [8], can be found in [2].

3 Advances in Grounding Programs with Functional Terms

In the following, we present a new version of the DLV grounder which is able to deal with ASP programs written in the language of Section 2, which has been conceived in order to improve the old one. The two system architectures are (at a very high level) depicted in Fig. 1.

3.1 Native Functional Terms vs Rewriting

As reported in [3], the support for functional terms has firstly been added to the state-of-the-art ASP system DLV [9] by means of a proper rewriting strategy. The resulting system, named DLV-COMPLEX [6], includes a rewriting module that removes all functional terms, introducing a number of instances of proper predefined built-in predicates.

[3] The system herein presented provides some simple built-in predicates, such as the comparative predicates equality, less-than, and greater-than $(=, <, >)$.

The grounding module is fed with a new program without functional terms: no actual change is then needed in the grounding process already implemented in DLV.

More in detail, any functional term $t = f(X_1, \ldots, X_n)$ appearing in some rule $r \in P$ is replaced by a fresh variable F, and one of the following atom is then added to $B(r)$:

- $\#function\_pack(f, X_1, \ldots, X_n, F)$ if either t appears in $H(r)$ or t appears in $B^-(r)$;
- $\#function\_unpack(F, f, X_1, \ldots, X_n)$ if t appears in $B^+(r)$.

Example 1. The rule: $p(f(f(X))) \text{ :- } q(X, g(X, Y)).$ will be rewritten as follow:

$$\begin{aligned} p(F_1) \text{ :- } & \#function\_pack(F_1, f, F_2), \#function\_pack(F_2, f, X), \\ & q(X, F_3), \#function\_unpack(F_3, g, X, Y). \end{aligned}$$

Note that rewriting the nested functional term $f(f(X))$ generates two instances of $\#function\_pack$ in the body: (i) for the inner f function having X as argument and (ii) for the outer f function having as argument the fresh variable F_2, representing the inner functional term. □

In practice, both $\#function\_pack$ and $\#function\_unpack$ built-in predicates act on strings; the former builds a new string concatenating a functor with its arguments, while the latter unfolds the functional term (again, represented by a string) identifying the functor and the list of its arguments, in order to provide values to the variables of the $\#function\_unpack$ built-in.

On the one hand, such rewriting strategy allowed, at the time of implementing, an effective support to functions without any drastic change in the grounding process; on the other hand, it suffers from many drawbacks, if efficiency is taken into account: apart from the rewriting time, which is negligible, the approach clearly pays a price while manipulating strings (storing, tokenizing, comparing, etc.), which grows more and more in case of programs that make a massive use of functional terms.

Experiences with such drop of performances pointed out the need for a re-engineering of the grounding process while dealing with functions. First of all, we redesigned the involved DLV internal data structure for managing functional terms as native data, in order to avoid expensive string manipulation. We roughly describe next the most relevant changes introduced into the grounder; going too much into implementation details is out of the scope of this work.

The DLV grounder originally supported three kind of terms: strings constants, integer constants, and variables; we introduced functions. For performance reasons, each term was internally associated with an integer value representing the index of the position in a proper table, in which its actual value is stored. Clearly, such a table were not suitable for storing a functional term, because of the need to store also the list of its arguments. Note that arguments of a function are also (possibly functional) terms: thus, a sort of recursive structure is needed. For representing functional terms, we exploited a new table storing functors, in order to associate each functor with an integer; then, an additional table (for functional terms) is used, where each entry is a pair of a functor index and a list of object terms representing its arguments (which, in turn, might be functional terms). More in detail, each object term features a flag telling its type (functional, variable, etc.) and an index; the table referred by the index depends on the flag. Note that, if the argument of a functional term is, again, functional, the index "recursively" refers to another entry in the functional term table. It is also worth noting that a functional term might be both a variable and a constant, but this makes no difference in terms of its representation. In particular, a functional term is considered as a variable if at least one argument in its list is a variable (or a variable functional term); it is treated as a constant otherwise.

The introduction of a new type of term has a strong impact on the whole grounding process: it requires relevant changes to many subtasks such as body reordering, indexing, backjumping, matching. However, it brings also significant performance improvements (see Section 4). In particular, the matching procedure takes great advantage from the possibility to use integers instead of strings while comparing functional terms.

3.2 Argument-Restricted vs Finite-Domain Programs

Since the grounding process for an ASP program with functional terms might not terminate [4], checking if termination can be “a priori” guaranteed is a useful feature in many cases.

The DLV-COMPLEX [3] system implemented a module for recognizing the membership to a class of programs that is known to be computable, namely the class of *finite-domain* programs, introduced in [2]. Other computable classes of ASP programs with functions have been identified from then on; in particular, the class of *argument-restricted* programs [10] significantly extends both the class of finite-domain and the class of λ-*restricted* [7] programs. According to [10], a program is argument-restricted if it has an argument ranking: basically, if it is possible to find a mapping from each argument of each predicate appearing in the program to an integer, such that a set of given conditions is satisfied. Such a condition is constrained for each variable in each predicate argument appearing in the head of each rule, and takes into account the head-body differences between nesting levels and values of argument ranking. Intuitively, it might be seen as a sort of stratification, but applied to argument predicates instead of atoms.

Example 2. The following program is not a finite-domain program, but it is argument-restricted.

$$p(f(X)) :- q(X). \qquad q(X) :- p(X), r(X). \qquad r(0). \qquad p(0).$$

Roughly (see [10] for a formal definition), let nl(var, atom) denote the nesting level of a variable in an atom; then, the inequalities to fulfill are:

$$rank(p[1]) - rank(q[1]) \geq nl(X, p(f(X))) - nl(X, q(X)) \qquad \text{(for the first rule)}$$
$$rank(q[1]) - rank(r[1]) \geq nl(X, q(X)) - nl(X, r(X)) \qquad \text{(for the second rule)}$$

An argument ranking exists in this case: indeed, the mapping:

$$rank(p[1]) = 1 \qquad rank(q[1]) = rank(r[1]) = 0$$

satisfies both inequalities above. □

Example 3. The following program is not argument-restricted.

$$p(f(X)) :- p(X). \qquad p(0).$$

For this program the inequality to be fulfilled is:

$$rank(p[1]) - rank(p[1]) \geq nl(X, p(f(X))) - nl(X, p(X))$$

but there cannot exist any argument ranking in this case, since every assignment for $rank(p[1])$ would fail (the inequality above becomes $0 \geq 1$). □

test	DLV-**rewriting**	DLV-**native**	**clingo**
cycles.1	313,3	51,0	308,1
cycles.2	225,1	96,7	187,6
cycles.3	26,6	5,0	76,7
cycles.4	19,0	6,5	134,2
cycles.5	20,1	8,7	166,6
hanoi.7discs.01	155,3	8,3	4,6
hanoi.7discs.02	194,7	10,0	5,4
hanoi.7discs.03	242,7	11,5	6,2
hanoi.7discs.04	7,5	0,4	0,2
hanoi.7discs.05	73,3	3,7	2,1
hanoi.7discs.06	98,0	4,5	2,6
hanoi.7discs.07	57,2	3,0	1,7
hanoi.7discs.08	82,5	4,3	2,5
hanoi.7discs.09	122,4	5,9	3,3
hanoi.7discs.10	87,4	4,8	2,7
palindromes.2	1031,3	1,3	1,1
palindromes.3	–	43,9	31,8
palindromes.4	–	16,9	10,5
palindromes.5	–	19,4	10,5
palindromes.6	–	200,9	98,5
paths.1	172,7	145,8	129,5
paths.2	172,7	145,6	129,4
paths.3	57,2	14,6	158,3
paths.4	15,2	11,7	220,8
paths.5	47,0	0,3	0,2

Table 1. Benchmark Results (times are reported in seconds)

The new prototype has been equipped with a proper module able to statically decide wether an input program is argument restricted, or not. The module, enabled by default and named "ar-checker", substitutes the "fd-checker" module of DLV-COMPLEX (Figure 1). If the user is confident that the program can be grounded in finite time (even if it does not belong to the class of argument-restricted programs), then she can disable the ar-checker; this can be done by specifying the command-line option `-nofinitecheck`. Another way for guaranteeing termination is the choice of a maximum allowed nesting level N for functional terms (command-line option `-maxnestinglevel=<N>`).

The implementation of the ar-checker exploits the idea depicted in [10]. Values for a ranking are computed as the least fixpoint of a monotone operator; since such a ranking, if it exists, cannot exceed an upper bound M (statically computable as the maximum nesting level times the total number of predicate arguments), our algorithm answers YES as soon as the fixpoint is reached, and NO in case some value becomes greater than M.

4 Experiments and Conclusions

We report in this Section the experimental activities we carried out with the aim to assess the impact of the new implementation approach on the DLV grounder performances. We start by briefly describing the benchmark problems considered.

Hanoi Towers. The classic Towers of Hanoi puzzle problem with three pegs and n disks of increasing size.
Paths in a Graph. Given a directed graph, find all paths of a maximum given length connecting each pair of nodes.
Cycles in a Graph. Given a directed graph, find all cycles involving at most a given number of nodes. A cycle has to be intended as a path (as in the previous problem) where the starting node and the ending node coincide.

Palindromic Words. Given a predefined set of chars, find all sequences of a given length that are equal to their reverse, and represent a meaningful word according to a given dictionary.

In order to properly test the system modules mainly involved in our work, we considered benchmark problems whose encodings heavily rely on function symbols, that is with many rules exploiting function symbols and involving the generation of a large number of ground atoms with functional terms. Moreover, for all benchmark the most part of computation time is spent in the grounding phase; in particular, all encodings but Hanoi Towers are normal and stratified. In order to meet space constraints, encodings are not shown here; for the sake of full reproducibility, encodings, instances and binary files can be found at a proper web link.[4] For each problem, we have randomly generated five instances; for hanoi towers, we took ten 7-disc instances from the second ASP Competition [5], and suitably adapted them in order to take advantage of functions.

The system tested are: DLV-*rewriting* (official DLV release 2010-10-14), wich consists of DLV featuring the rewriting strategy; DLV-*native* (the new developed prototype), that features native function terms; and *clingo* (version 3.0.3), a very efficient ASP solver [1]. For every instance, we allowed a maximum running (CPU) time of 900 seconds (15 minutes).

The machine exploited, featuring two Intel Xeon “Woodcrest” (quad-core) processors with 4MB of L2 cache, equipped with 4GB of RAM, runs GNU Linux Debian 4.1.1-21 (kernel 2.6.23.9 amd64 smp).

Results are reported in Table 1; a dash mark (“–”) means that a system was not able to solve that instance within the allowed amount of time. In general, results confirm the intuition that the integration of function terms as native data, thus avoiding the rewriting phase involving heavy string manipulation, has a very positive impact on the efficiency of DLV. In particular, the new approach regularly outperforms the rewriting-based one. In many cases we noticed impressive speedups: for instance, DLV-rewriting was able to solve just one out of five instances of palindromic words, while DLV-native solved them all (in less than one minute in most cases).

Notably, the new approach allows DLV to bridge the performance gap with clingo, when a heavy usage of functional terms is required.

References

1. `clingo` homepage, `http://potassco.sourceforge.net/`
2. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008). Lecture Notes in Computer Science, vol. 5366, pp. 407–424. Springer, Udine, Italy (Dec 2008)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An ASP System with Functions, Lists, and Sets. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009). Lecture Notes in Computer Science, vol. 5753, pp. 483–489. Springer Verlag (Sep 2009)
4. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
5. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The Second Answer Set Programming Competition. In: Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 5753, pp. 637–654. Springer Berlin / Heidelberg (2009)

[4] `http://www.mat.unical.it/calimeri/downloads/gttv-2011-calimeri-etal.zip`

6. Faber, W., Pfeifer, G.: `DLV` homepage (since 1996), `http://www.dlvsystem.com/`
7. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07. Lecture Notes in Computer Science, vol. 4483, pp. 266–271. Springer Verlag, Tempe, Arizona (May 2007)
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)
10. Lierler, Y., Lifschitz, V.: One More Decidable Class of Finitely Ground Programs. In: Proceedings of the 25th International Conference on Logic Programming (ICLP 2009). Lecture Notes in Computer Science, vol. 5649, pp. 489–493. Springer, Pasadena, CA, USA (Jul 2009)