Galliwasp: A Goal-Directed Answer Set Solver

Kyle Marple and Gopal Gupta

University of Texas at Dallas
800 W. Campbell Road
Richardson, TX 75080, USA

Abstract. *Galliwasp* is a goal-directed implementation of answer set programming. Unlike other answer set solvers, *Galliwasp* computes partial answer sets which are provably extensible to full answer sets. *Galliwasp* can execute arbitrary answer set programs in a top-down manner similar to SLD resolution. *Galliwasp* generates candidate answer sets by executing *ordinary rules* in a top-down, goal-directed manner using *coinduction*. *Galliwasp* next checks if the candidate answer sets are consistent with restrictions imposed by *OLON rules*. Those that are consistent are reported as solutions. Execution efficiency is significantly improved by performing the consistency check incrementally, i.e., as soon as an element of the candidate answer set is generated. We discuss the design of the *Galliwasp* system and its implementation. *Galliwasp*'s performance figures, which are comparable to other popular answer set solvers, are also presented.

1 Introduction

As answer set programming (ASP) [8] has gained popularity, answer set solvers have been implemented using various techniques. These techniques range from simple guess-and-check based methods to those based on SAT solvers and complex heuristics. None of these techniques impart any operational semantics to the answer set program in the manner that SLD resolution does to Prolog; they can be thought of as being similar to bottom-up methods for evaluating logic programs. Earlier we described [10,17] how to find partial answer sets by means of a top-down, goal-directed method based on coinduction. The *Galliwasp* system [16], described in this paper, is the first efficient implementation of this goal-directed method. *Galliwasp* is consistent with the conventional semantics of ASP: no restrictions are placed on the queries and programs that can be executed. That is, any A-Prolog program can be executed on *Galliwasp*.

Galliwasp is an implementation of A-Prolog [7] that uses grounded normal logic programs as input. The underlying algorithm of *Galliwasp* leverages coinduction [23,10] to find partial answer sets containing a given query. Programs are executed in a top-down manner in the style of Prolog, computing partial answer sets through SLD-style call resolution and backtracking. Each partial answer set is provably extensible to a complete answer set [10,17].

The algorithm of *Galliwasp* uses call graphs to classify rules according to two attributes: (i) if a rule can be called recursively with an odd number of negations between the initial call and its recursive invocation, it is said to contain an *odd loop over negation* (OLON) and referred to as an OLON rule for brevity, and (ii) if a rule has at least one path in the call graph that will not result in such a call, it is called an ordinary rule. Our goal-directed method uses ordinary rules to generate candidate answer sets (via co-induction extended with negation) and OLON rules to reject invalid candidate sets. The procedure can be thought of as following the *generate and test* paradigm with ordinary rules being used for generation of candidate answer sets and OLON rules for testing that a candidate answer set is, indeed, a valid answer set.

The main contribution of this paper is an execution mechanism in which consistency checks are incrementally executed as soon as an element of the candidate answer set is generated. By interleaving the generation and testing of candidate answer sets in this manner, *Galliwasp* speeds up execution significantly. As soon as an element `p` is added to a candidate answer set, OLON rules that might reject the answer set due to presence of `p` in it are invoked.

In this paper we describe the design and implementation of *Galliwasp*. This includes an overview of goal-directed ASP, the design and implementation of *Galliwasp*'s components, and techniques developed to improve performance. Finally, the performance figures are presented and compared to other ASP solvers.

There are many advantages of a goal-directed execution strategy: (i) ASP can be extended to general predicates [18], i.e., to answer set programs that do not admit a finite grounding; (ii) extensions of ASP to constraints (in the style of CLP(R)) [12], probabilities (in the style of ProbLog [4]), etc., can be elegantly realized; (iii) Or-parallel implementations of ASP that leverage techniques from parallel Prolog implementations [11] can be developed; (iv) abductive reasoning [14] can be incorporated with greater ease. Work is in progress to extend *Galliwasp* in these directions.

2 Goal-Directed Answer Set Programming

The core of *Galliwasp* is a goal-directed execution method for computing answer sets which is sound and complete with respect to the Gelfond-Lifschitz method [10,17]. For convenience, two key aspects of our goal-directed method are summarized here: its handling of OLON rules and its use of coinduction.

2.1 OLON Rules and Ordinary Rules

Our goal-directed method requires that we identify and properly handle both ordinary and OLON rules, as defined in the introduction. Rules are classified by constructing and examining a call graph. Unlike in Prolog, literals can be either positive or negative, and the attributes of a rule are determined by the number of negations between any recursive calls in the call graph.

To build the call graph, each positive literal is treated as a node in the graph. For each rule, arcs are drawn from the node corresponding to the head of the rule to every node corresponding to the positive form of a literal in the body of the rule. If the literal in the body is positive, the arc is blue; if the literal is negative, the arc is red. At most one of each color arc is created between any two nodes, and each arc is labelled to identify all of the rules associated with it.

When the graph has been created, each path is examined in a bottom-up fashion, stopping when any literal in the path is repeated. If there is a path from a node `N` to itself such that the number of red arcs on the path is odd, then we call all of the rules for N's literal associated with the path OLON rules. If there is no path from a node `N` to itself, or if there is a path from `N` to itself such that the number of red arcs on the path is even, we call all of the rules for N's literal associated with the path ordinary rules.

Every rule will have at least one of the two attributes. Additionally, a rule can be both an OLON rule and an ordinary rule, as the example below illustrates:

```
    p :- not q. ... (i)
    q :- not p. ... (ii)
    q :- not r. ... (iii)
    r :- not p. ... (iv)
```

Rule (i) is both an OLON rule and an ordinary rule: ordinary because of rule (ii), OLON because of rules (iii) and (iv).

2.2 Coinductive Execution

The goal-directed method [10,17] generates candidates answer sets by executing ordinary rules via coinduction extended with negation. Given a query, an extended query is constructed to enforce the OLON rules. This extended query is executed in the style of SLD resolution with coinduction (co-SLD resolution) [23,10].

Under the operational semantics of coinduction, a call `p` succeeds if it unifies with one of its ancestor calls. Each call is remembered, and this set of ancestor calls forms the *coinductive hypothesis set* (CHS). Under our algorithm, the CHS also constitutes a candidate answer set. As such, it would be inconsistent for `p` and `not p` to be in the CHS at the same time: when this is about to happen, the system must backtrack. As a result, only ordinary rules can generate candidate answer sets. Any OLON rule that is not also an ordinary rule will fail during normal execution, as it will attempt to add the negation of its head to the CHS.

Candidate sets produced by ordinary rules must still be checked for consistency with the OLON rules in a program. To accomplish this, the query is extended to perform this check.

Before looking at this extension, let us consider an OLON rule of the form

```
    p :- B, not p.
```

where `B` is a conjunction of goals. One of two cases must hold for an answer set to exist: (i) `p` is in the answer set through another rule in the program, (ii) at least one goal in `B` must fail. This is equivalent to saying that the negation of the rule, `not B` $\vee$ `p` must succeed. Note that checks of this form can be created for any OLON rule [10,17]. This includes both OLON rules of the form

```
p :- B.
```

where the call to `not p` is indirect, and headless rules of the form

```
:- B.
```

where the negation `not B` must succeed.

Our method handles OLON rules by creating a special rule, called the NMR check, which contains a sub-check for each OLON rule in a program. The body of the NMR check is appended to each query prior to evaluation, ensuring that any answer set returned is consistent with all of the OLON rules in the program. Prior to creating the sub-checks, a copy of each OLON rule is written in the form

```
p :- B, not p.
```

as shown above, with the negation of the rule head added to the body in cases where the call is indirect. Each sub-check is then created by negating the body of one of the copied rules. For instance, the sub-check for a rule

```
p :- q, not p.
```

will be

```
chk_p :- not q.
chk_p :- p.
```

The body of the NMR check will then contain the goal `chk_p`, ensuring that the OLON rule is properly applied.

Some modification to co-SLD resolution is necessary to remain faithful to ASP's stable model semantics. This is due to the fact that coinduction computes the greatest fixed point of a program, while the GL method computes a fixed point that is between the least fixed point (*lfp*) and greatest fixed point (*gfp*) of a program, namely, the *lfp* of the residual program.To adapt coinduction for use with ASP, our goal-directed method restricts coinductive success to cases in which there are an even, non-zero number of calls to `not` between a call and an ancestor to which it unifies. This prevents rules such as

```
p :- p.
```

from succeeding, as this would be the case under normal co-SLD resolution. To compute an answer set for a given program and query, our method performs co-SLD resolution using this modified criterion for coinductive success. When both a query and the NMR check have succeeded, the CHS will be a valid answer set [10,17]. Consider the program below:

```
p :- q.
q :- p.
```

Under normal co-SLD resolution, both `p` and `q` would be allowed to succeed, resulting in the incorrect answer set {`p, q`}. However, using our modified criterion, both rules will fail, yielding the correct answer set {`not p, not q`}.

3 The Galliwasp System

The *Galliwasp* system is an implementation of the above SLD resolution-style, goal-directed method of executing answer set programs. A number of issues arise that are discussed next. Improvements to the execution algorithm that make *Galliwasp* more efficient are also discussed.

3.1 Order of Rules and Goals

As with normal SLD resolution, rules for a given literal are executed in the order given in the program, with the body of each rule executed left to right. With the exception of OLON rules, once the body of a rule whose head matches a given literal has succeeded, the remaining rules do not need to be accessed unless failure and backtracking force them to be selected.

As in top-down implementations of Prolog, this means that the ordering of clauses and goals can directly impact the runtime performance of a program. In the best case scenario, this can allow *Galliwasp* to compute answers much faster than other ASP solvers, but in the worst case scenario *Galliwasp* may end up backtracking significantly more, and as a result take much longer.

One example of this can be found in an instance of the Schur Numbers benchmark used in Sect. 5. Consider the following clauses from the grounded instance for 3 partitions and 13 numbers:

```
_false :- not haspart(1).
_false :- not haspart(2).
_false :- not haspart(3).
_false :- not haspart(4).
_false :- not haspart(5).
_false :- not haspart(6).
_false :- not haspart(7).
_false :- not haspart(8).
_false :- not haspart(9).
_false :- not haspart(10).
_false :- not haspart(11).
_false :- not haspart(12).
_false :- not haspart(13).
```

During benchmarking, *Galliwasp* was able to find an answer set for the program containing the above clauses in 0.2 seconds. However, the same program with the order of only the above clauses reversed could be left running for several minutes without terminating.

Given that *Galliwasp* is currently limited to programs that have a finite grounding, its performance can be impacted by the grounder program that is used. In the example above, all 13 ground clauses are generated by the grounding of a single rule in the original program. Work is in progress to extend *Galliwasp* to allow direct execution of datalog-like ASP programs (i.e., those with only constants and variables) without grounding them first. In such a case, the user would have much more control over the order in which rules are tried.

3.2 Improving Execution Efficiency

The goal-directed method described in Sect. 2 can be viewed as following the *generate and test* paradigm. Given a query `Q, N` where `Q` represents the original user query, and `N` the NMR check, the goal directed procedure *generates* candidate answer sets through the execution of `Q` using ordinary rules. These candidate answer sets are *tested* by the NMR check `N` which rejects a candidate if the restrictions encoded in the OLON rules are violated.

Naive generate and test can be very inefficient, as the query `Q` may generate a large number of candidate answer sets, most of which may be rejected by the NMR check. This can lead to a significant amount of backtracking, slowing down the execution. Execution can be made significantly more efficient by generating and testing answer sets incrementally. This is done by interleaving the execution of `Q` and `N`. As soon as a literal, say `p`, is added to a candidate answer set, goals in NMR check that correspond to OLON rules that `p` may violate are invoked. We illustrate the incremental generate and test algorithm implemented in *Galliwasp*.

Consider the rules below:

```
p :- r.
p :- s.
q :- big_goal.
r :- not s.
s :- not r.
:- p, r.
```

Let us assume that this fragment is part of a program where `big_goal` will always succeed, but is computationally expensive and contains numerous choice points. For simplicity, let us also assume that the complete program contains no additional OLON rules, and no additional rules for the literals `p`, `q`, `r` or `s`. As the program contains only the OLON rule from the fragment above, the NMR check will have only one sub-check, which will consist of two clauses:

```
nmr_check :- chk_1.
chk_1 :- not p.
chk_1 :- not r.
```

Next, assume that the program is compiled and run with the following query:

```
?- p, q.
```

As with any query, the NMR check will be appended, resulting in the final query:

```
?- p, q, nmr_check.
```

The NMR check ensures that `p` and `r` are never present in the same answer set, so any valid answer set will contain `p`, `q`, `not r`, `s`, `big_goal`, and the literals upon which `big_goal` depends.

If the program is run without NMR check reordering, the first clause for `p` will initially succeed, adding `r` to the CHS. Failure will not occur until the NMR check is reached, at which point backtracking will ensue. Eventually, the initial call to `p` will be reached, the second clause will be selected, and the query will eventually succeed. However, the choicepoints in `big_goal` could result in a massive amount of backtracking, significantly delaying termination. Additionally, `big_goal` will have to succeed twice before a valid answer set is found.

Now let us consider the same program and query using incremental generate and test. The first clause for p will still initially succeed. However, as r succeeds, the clause of chk_1 calling not r will be removed. Since only one clause will remain for chk_1, chk_1 will be reordered, placing it immediately after the call to p and before the call to q. The call to p will then succeed as before, but failure and backtracking will occur almost immediately, as the call to chk_1 will immediately call not p. As before, the call to not p will result in the second clause for p being selected and the query will eventually succeed. However, as failure occurs before big_goal is reached, the resulting backtracking is almost eliminated. Additionally, big_goal does not need to be called twice before a solution is found. As a result, execution with NMR check reordering will terminate much sooner than execution without reordering.

Adding incremental test and generation to *Galliwasp* leads to a considerable improvement in efficiency, resulting in the performance of the *Galliwasp* system becoming comparable to state-of-the-art solvers.

4 System Architecture of Galliwasp

The *Galliwasp* system consists of two components: a compiler and an interpreter. The compiler reads in a grounded instance of an ASP program and produces a compiled program, which is then executed by the interpreter to compute answer sets. An overview of the system architecture is shown in Fig. 4.

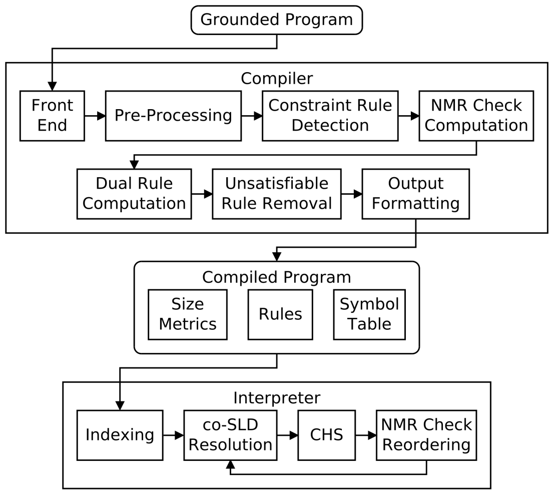

Fig. 1. *Galliwasp*'s system architecture.

4.1 Compiler

The primary goal of *Galliwasp*'s design is to maximize the runtime performance of the interpreter. Because *Galliwasp* computes answer sets for a program based on user-supplied queries, answer sets cannot be computed until a program is actually run by the interpreter. However, many of the steps in our goal-directed algorithm, as well as parsing, program transformations and static analysis, are independent of the query. The purpose of *Galliwasp*'s compiler is to perform as much of the query-independent work as possible, leaving only the actual computation of answer sets to the interpreter. Because a single program can be run multiple times with different queries, compilation also eliminates the need to repeat the query-independent steps every time a program is executed. The most important aspects of the compiler are the input language accepted, the parsing and pre-processing performed, the construction of the NMR check, the program transformations applied, and the formatting of the compiled output.

Input Language The *Galliwasp* compiler's input language is grounded A-Prolog [7], extended to allow partial compatibility with text-mode output of the *lparse* grounder. This allows grounded program instances to be obtained by invoking *lparse* with the `-t` switch, which formats the output as text.

Only a subset of *lparse*'s text output is supported by *Galliwasp*. The support is made possible by special handling of rules with the literal `_false` as their head, a convention used by *lparse* to add a head to otherwise headless rules, and support for *lparse*'s `compute` statement. Other features of *lparse*, such as constraint and weight literals, choice rules, and optimize statements, are not currently supported in *Galliwasp*. Work is in progress to support them.

When *lparse* encounters a headless rule, it produces a grounded rule with `_false` as the head and a compute statement containing the literal `not _false`. Because the literal `_false` is not present in the body of any rule, special handling is required to properly detect such rules as OLON rules.

The compute statements used by *lparse* are of the form

```
compute N { Q }.
```

where `N` specifies the number of answer sets to compute and `Q` is a set of literals that must be present in a valid answer set. Our system handles these statements by treating them as optional, hard-coded queries. If a compute statement is present, the interpreter may be run without user interaction, computing up to `N` answer sets using `Q` as the query. When the interpreter is run interactively, it ignores compute statements and executes queries entered by the user.

Parsing and Pre-Processing The compiler's front end and pre-processing stages prepare the input program for easy access and manipulation during the rest of the compilation process. The front end encompasses the initial lexical analysis and parsing of the input program, while the pre-processing stage handles additional formatting and simplification, and substitutes integers for literals.

After lexical analysis of the input program is performed, it is parsed into a list of rules and statements by a definite clause grammar (DCG). During parsing, a counter is used to number the rules as they are read, so that the relative order of rules for a given literal can be maintained.

The list of statements produced by the DCG is next converted into a list of rules and a single compute statement. Each rule is checked to remove duplicate goals. Any rule containing its head as a goal with no intervening negation will also be removed at this stage. The cases covered in this step should not normally occur, but as they are allowed by the language, they are addressed before moving on.

The next stage of pre-processing is integer substitution. Every propositional symbol `p` is mapped to a unique integer $N_p > 0$. A positive literal `p` is represented by N_p, while a negative literal `not p` is represented by $-N_p$. Since the interpreter must use the original names of propositions when it prints the answer, a table that maps each N_p to `p` is included in the compiled output.

Finally, the list of rules is sorted by head, maintaining the relative order of rules with the same head. This eliminates the need for repeated searching in subsequent stages of compilation. After sorting, compilation moves on to the next stage, the detection of OLON rules and construction of the NMR check.

Construction of the NMR Check Construction of the NMR check begins with the detection of the OLON rules in the ASP program. This detection is performed by building and traversing a call graph similar to the one described in Sect. 2.1. These rules are then used to construct the individual checks that form the NMR check, as described in Sect. 2.2.

Clauses for the sub-checks are treated as any other rule in the program, and subject to program transformation in the next stage of compilation. However, the NMR check itself is not modified by the program transformation stage. Instead, if the modified sub-checks allow for immediate failure, this will be detected at runtime by the interpreter.

Program Transformation The program transformation stage consists of computing dual rules, explained below, and removing rules when it can be trivially determined at compile time that they will never succeed. This stage of compilation improves performance without affecting the correctness of our algorithm.

Dual rules, i.e., rules for the negation of each literal, are computed as follows. For proposition `p` defined by the rules,

```
    p :- B₁.
    ...
    p :- Bₙ.
```

where each B_i is a conjunction of positive and negative literals, its dual

```
    not p :- not B₁, ..., not Bₙ.
```

is added to the program. For any proposition `q` for which there are no rules whose head contains `q`, a fact is added for `not q`.

The addition of dual rules simplifies the design of the interpreter by removing the need to track the scope of negations: all rules can be executed in a uniform fashion, regardless of the number of negations encountered. When a negated calls are encountered, they are expanded using dual rules. While adding these rules may significantly increase the size of the program, this is not a problem: the interpreter performs indexing that allows it to access rules in constant time.

After the dual rules have been computed, the list is checked once more to remove simple cases of rules that can be trivially determined to always fail. This step simply checks to see if a fact exists for the negation of some goal in the rule body and removes the rule if this is the case. If the last rule for a literal is removed, a fact for the negation is added and subsequent rules calling the literal will also be removed.

Output Formatting As with the rest of the compiler, the output produced by the compiler is designed to reduce the amount of work performed by the interpreter. This is done by including space requirements and sorting the rules by their heads before writing them to the output.

As a result of the output formatting, the interpreter is able to read in the input and create the necessary indexes in linear time with respect to the size of the compiled program. After indexing, all that remains is to execute the program, with all other work having been performed during compilation.

4.2 Interpreter

While the compiler was designed to perform a variety of time consuming tasks, the interpreter has been designed to maximize run-time performance, finding an answer set or determining failure as quickly as possible. Two modes of operation, interactive and automatic, are supported. When a program is run interactively, the user can input queries and accept or reject answer sets as can be done with answers to a query in Prolog. In automatic mode it executes a compute statement that is included in the program. In either mode, the key operations can be broken up into three categories: program representation and indexing, co-SLD resolution, and dynamic incremental enforcement of the NMR check.

Program Representation and Indexing One of the keys to *Galliwasp*'s performance is the indexing performed prior to execution of the program. As a result, look-up operations during execution can be performed in constant time, much as in any implementation of Prolog.

As mentioned in Sect. 4.1, the format of the compiler's output allows the indexes to be created in time that is linear with respect to the size of the program. The compiled program includes the number of positive literals, which allows a hash table to be constructed using an extremely simple perfect hash function. For n positive literals, $2n$ entries are needed, as 0 is never used. For a literal L which may be positive or negative,

$$index(L) = \begin{cases} L & L > 0, \\ 2n + 1 + L & L < 0 \end{cases}$$

Because the program is sorted by the compiler, the main index can be created in a single pass, using fixed length arrays rather than dynamically allocated memory.

To allow for the NMR check interleaving and simplification discussed in Sect. 4.2, the query and NMR check are stored separately from the rest of the program. Whereas the rules of the program, including the sub-checks of the NMR check, are stored in ordinary arrays, the query and NMR check are stored in a linked list. This allows their goals to be reordered in constant time. Additional indexing is also performed, linking each literal to its occurrences in the NMR check. Together, these steps allow modification of the NMR check with respect to a given literal to be performed in linear time with respect to the number times the literal and its dual occurs in the sub-checks.

Co-SLD Resolution Once the program has been indexed, answer sets are found by executing the query using coinduction, as described in Sect. 2.2. Each call encountered is checked against the CHS. If the call is not present, it is added to the CHS and expanded according to ordinary SLD resolution. If the call is already in the CHS, immediate success or failure occurs, as explained below.

As mentioned in Sect. 2.2, our algorithm requires that a call that unifies with an ancestor cannot succeed coinductively unless it is separated from that ancestor by an even, non-zero number of intervening negations. This is facilitated by our use of dual rules, discussed in Sect. 4.1.

When the current call is of the form `not p` and the CHS contains `p` (or vice versa), the current call must fail, lest the CHS become inconsistent. If the current call is identical to one of the elements of the CHS, then the number of intervening negations must have been even. However, it is not clear that it was also non-zero. This additional information is provided by a counter that tracks of the number of negations encountered between the root of the proof tree and the tip of the current branch. When a literal is added to the CHS, it is stored with the current value of the counter. A recursive call to the literal can coinductively succeed only if the counter has increased since the literal was stored.

Dynamic Incremental Enforcement of the NMR Check Because recursive calls are never expanded, eventual termination is guaranteed, just as for any other ASP solver. However, since we use backtracking, the size of the search space can cause the program to execute for an unreasonable amount of time.

To alleviate this problem, we introduced the technique of incrementally enforcing the NMR check (cf. Sect. 3.2 above). Our technique can be viewed as a form of coroutining: steps of the testing and generation phases are interleaved. The technique consists of simplifying NMR sub-checks and of reordering the calls to sub-checks within the query. These modifications are done whenever a literal is added to the CHS, and are "undone" upon backtracking.

When a literal is added to the CHS, all occurrences of that literal are removed from every sub-check, as they would immediately succeed when treated as calls.

If such a removal causes the body of a sub-check clause to become empty, the entire sub-check is removed, as it is now known to be satisfied.

The next step is to remove every sub-check clause that contains the negation of the literal: such a clause cannot be satisfied now. If this clause causes the entire sub-check to disappear, the literal obviously cannot be added to the CHS, and backtracking occurs. If the sub-check does not disappear, but is reduced to a single clause (i.e., it becomes deterministic), then a call to the sub-check is moved to the front of the current query: this ensures early testing of whether the addition of the literal is consistent with the relevant OLON rules. If the resultant sub-check contains more than one clause, there is no attempt to execute it earlier than usual, as that might increase the size of the search space.

As mentioned in Sect. 4.2, indexing allows these modifications to be performed efficiently. If the original search space is small, they may result in a small increase in runtime. In most non-trivial cases, however, the effect is a dramatic decrease in the size of the search space. It is this technique that enables *Galliwasp* to be as efficient as illustrated in the next section: early versions of the system used a simple generate-and-test approach, but many of the programs that now terminate in a fraction of a second ran for inordinate amounts of time.

5 Performance Results

In this section, we compare the performance of *Galliwasp* to that of *clasp* [5], *cmodels* [9] and *smodels* [20] using instances of several problems. The range of problems is limited somewhat by the availability of compatible programs. While projects such as Asparagus [2] have a wide variety of ASP problem instances available, the majority use *lparse* features unsupported by *Galliwasp*. Work is in progress to support these features.

The times for *Galliwasp* in Table 1 are for the interpreter reading compiled versions of each problem instance. No queries are given, so the NMR check alone is used to find solutions. The times for the remaining solvers are for the solver reading problem instances from files in the *smodels* input language. A timeout of 600 seconds was used, with the processes being automatically killed after that time and the result being recorded as N/A in the table.

Galliwasp is outperformed by *clasp* in all but one case and outperformed by *cmodels*, in most cases, but the results are usually comparable. *Galliwasp*'s performance can vary significantly, even between instances of the same problem, depending on the amount of backtracking required. In the case of programs that timed out or took longer than a few seconds, the size and structure of the programs simply resulted in a massive amount of backtracking.

We believe that manually reordering clauses or changing the grounding method will improve performance in most cases. In particular, experimentation with manual clause reordering has resulted in execution times on par with *clasp*'s. However, the technique makes performance dependent on the query used to determine the optimal ordering. As a result, the technique is not general enough

Table 1. Performance results; times in seconds

Problem	*Galliwasp*	*clasp*	*cmodels*	*smodels*
queens-12	0.033	0.019	0.055	0.112
queens-13	0.034	0.022	0.071	0.132
queens-14	0.076	0.029	0.098	0.362
queens-15	0.071	0.034	0.119	0.592
queens-16	0.293	0.043	0.138	1.356
queens-17	0.198	0.049	0.176	4.293
queens-18	1.239	0.059	0.224	8.653
queens-19	0.148	0.070	0.272	3.288
queens-20	6.744	0.084	0.316	47.782
queens-21	0.420	0.104	0.398	95.710
queens-22	69.224	0.112	0.472	N/A
queens-23	1.282	0.132	0.582	N/A
queens-24	19.916	0.152	0.602	N/A
mapclr-4x20	0.018	0.006	0.011	0.013
mapclr-4x25	0.021	0.007	0.014	0.016
mapclr-4x29	0.023	0.008	0.016	0.018
mapclr-4x30	0.026	0.008	0.016	0.019
mapclr-3x20	0.022	0.005	0.009	0.008
mapclr-3x25	0.065	0.006	0.011	0.010
mapclr-3x29	0.394	0.006	0.012	0.011
mapclr-3x30	0.342	0.007	0.012	0.011
schur-3x13	0.209	0.006	0.010	0.009
schur-2x13	0.019	0.005	0.007	0.006
schur-4x44	N/A	0.230	1.394	0.349
schur-3x44	7.010	0.026	0.100	0.076
pigeon-10x10	0.020	0.009	0.020	0.025
pigeon-20x20	0.050	0.048	0.163	0.517
pigeon-30x30	0.132	0.178	0.691	4.985
pigeon-8x7	0.123	0.072	0.089	0.535
pigeon-9x8	0.888	0.528	0.569	4.713
pigeon-10x9	8.339	4.590	2.417	46.208
pigeon-11x10	90.082	40.182	102.694	N/A

for use in performance comparisons, and no reordering of clauses was used to obtain the results reported here.

6 Related and Future Work

Galliwasp's goal-directed execution method is based on a previously published technique, but has been significantly refined [10,19]. In particular, the original algorithm was limited to ASP programs which were call-consistent or order-consistent [19]. Additionally, the implementation of the previous algorithm was written as a Prolog meta-interpreter, and incapable of providing results comparable to those of existing ASP solvers. *Galliwasp*, written in C, is the first goal-directed implementation capable of direct comparison with other ASP solvers.

Various other attempts have been made to introduce goal-directed execution to ASP. However, many of these methods rely on modifying the semantics or restricting the programs and queries which can be used, while *Galliwasp*'s algorithm remains consistent with stable model semantics and works with any arbitrary program or query. For example, the revised Stable Model Semantics [21] allows goal-directed execution [22], but does so by modifying the stable model semantics underlying ASP. SLD resolution has also been extended to ASP through credulous resolution [3], but with restrictions on the type of programs and queries allowed. Similar work may also be found in the use of ASP with respect to abduction [1], argumentation [13] and tableau calculi [6].

Our plans for future work focus on improving the performance of *Galliwasp* and extending its functionality. In both cases, we are planning to investigate several possible routes.

With respect to performance, we are looking into exercising better control over the grounding of programs, or eliminating of grounding altogether. The development of a grounder optimized for goal-directed execution is being investigated ([15], forthcoming), as is the extension of answer set programming to predicates (preliminary work can be found in [18]). Both have the potential to significantly improve *Galliwasp*'s performance by reducing the amount of backtracking that results from the use of grounded instances produced using *lparse*.

In the area of functionality, we plan to expand *Galliwasp*'s input language to support features commonly allowed by other solvers, such as constraint literals, weight literals and choice rules. Extensions of *Galliwasp* to incorporate constraint logic programming, probabilistic reasoning, abduction and or-parallelism are also being investigated. We believe that these extensions can be incorporated more naturally into *Galliwasp* given its goal-directed execution strategy.

The complete source for *Galliwasp* is available from the authors at [16].

7 Conclusion

In this paper we introduced the goal-directed ASP solver *Galliwasp* and presented *Galliwasp*'s underlying algorithm, limitations, design, implementation and performance. *Galliwasp* is the first approach toward solving answer sets that uses goal-directed execution for general answer set programs. Its performance is comparable to other state-of-the-art ASP solvers. *Galliwasp* demonstrates the viability of the top-down technique. Goal-directed execution can offer significant potential benefits. In particular, *Galliwasp*'s underlying algorithm paves the way for ASP over predicates [18] as well as integration with other areas of logic programming. Unlike in other ASP solvers, performance depends on the amount of backtracking required by a program: at least to some extent this can be controlled by the programmer. Future work will focus on improving performance and expanding functionality.

Acknowledgments: The authors are grateful to Feliks Kluźniak for many discussions and ideas.

References

1. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. Theory and Practice of Logic Programming 4, 383–428 (July 2004)
2. Asparagus: `http://asparagus.cs.uni-potsdam.de` (2012)
3. Bonatti, P.A., Pontelli, E., Son, T.C.: Credulous Resolution for Answer Set Programming. In: Proceedings of the 23rd national conference on Artificial Intelligence - Volume 1. pp. 418–423. AAAI'08, AAAI Press (2008)
4. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: Proceedings of the 20th international joint conference on Artifical Intelligence. pp. 2468–2473. IJCAI'07, Morgan Kaufmann (2007)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Clasp: A Conflict-Driven Answer Set Solver. In: Proceedings of the 9th international conference on Logic Programming and Nonmonotonic Reasoning. pp. 260–265. LPNMR'07, Springer-Verlag (2007)
6. Gebser, M., Schaub, T.: Tableau Calculi for Answer Set Programming. In: Proceedings of the 22nd international conference on Logic Programming. pp. 11–25. ICLP'06, Springer-Verlag (2006)
7. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. pp. 413–451. Springer-Verlag (2002)
8. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Proceedings of the Fifth international conference on Logic Programming. pp. 1070–1080. MIT Press (1988)
9. Giunchiglia, E., Lierler, Y., Maratea, M.: SAT-Based Answer Set Programming. In: Proceedings of the 19th national conference on Artifical Intelligence. pp. 61–66. AAAI'04, AAAI Press (2004)
10. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications. In: Proceedings of the 23rd international conference on Logic Programming. pp. 27–44. ICLP'07, Springer-Verlag (2007)
11. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel Execution of Prolog Programs: A Survey. ACM Transactions on Programming Languages and Systems 23(4), 472–602 (July 2001)

12. Jaffar, J., Lassez, J.L.: Constraint Logic Programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 111–119. POPL'87, ACM (1987)
13. Kakas, A., Toni, F.: Computing Argumentation in Logic Programming. Journal of Logic and Computation 9(4), 515–562 (1999)
14. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1992)
15. Marple, K.: Design and Implementation of a Goal-directed Answer Set Programming System. Ph.D. thesis, University of Texas at Dallas
16. Marple, K.: galliwasp. http://www.utdallas.edu/~kbm072000/galliwasp/
17. Marple, K., Bansal, A., Min, R., Gupta, G.: Goal-Directed Execution of Answer Set Programs. Tech. rep., University of Texas at Dallas (2012), http://www.utdallas.edu/~kbm072000/galliwasp/publications/goaldir.pdf
18. Min, R.: Predicate Answer Set Programming with Coinduction. Ph.D. thesis, University of Texas at Dallas (2010)
19. Min, R., Bansal, A., Gupta, G.: Towards Predicate Answer Set Programming via Coinductive Logic Programming. In: AIAI. pp. 499–508. Springer (2009)
20. Niemelä, I., Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In: Logic Programming And Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 1265, pp. 420–429. Springer-Verlag (1997)
21. Pereira, L., Pinto, A.: Revised Stable Models - A Semantics for Logic Programs. In: Progress in Artificial Intelligence, Lecture Notes in Computer Science, vol. 3808, pp. 29–42. Springer-Verlag (2005)
22. Pereira, L., Pinto, A.: Layered Models Top-Down Querying of Normal Logic Programs. In: Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 5418, pp. 254–268. Springer-Verlag (2009)
23. Simon, L.: Extending Logic Programming with Coinduction. Ph.D. thesis, University of Texas at Dallas (2006)