Goal-Directed Execution of Answer Set Programs

Kyle Marple
University of Texas at Dallas
800 W. Campbell Road
Richardson, TX, USA
kbm072000@utdallas.edu

Ajay Bansal
Arizona State University
7231 E. Sonoran Arroyo Mall
Mesa, Arizona, USA
Ajay.Bansal@asu.edu

Richard Min
University of Texas at Dallas
800 W. Campbell Road
Richardson, TX, USA
min75243@hotmail.com

Gopal Gupta
University of Texas at Dallas
800 W. Campbell Road
Richardson, TX, USA
gupta@utdallas.edu

Abstract

Answer Set Programming (ASP) represents an elegant way of introducing non-monotonic reasoning into logic programming. ASP has gained popularity due to its applications to planning, default reasoning and other areas of AI. However, none of the approaches and current implementations for ASP are *goal-directed*. In this paper we present a technique based on *coinduction* that can be employed to design SLD resolution-style, goal-directed methods for executing answer set programs. We also discuss advantages and applications of such goal-directed execution of answer set programs, and report results from our implementation.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming

General Terms Algorithms

Keywords Answer set programming, goal-directed execution, coinduction

1. Introduction

Answer Set Programming (ASP) is an elegant way of developing non-monotonic reasoning applications. ASP has gained wide acceptance, and considerable research has been done in developing the paradigm as well as its implementations and applications. ASP has been applied to important areas such as planning, scheduling, default reasoning, reasoning about actions [3], etc. Numerous implementations of ASP have been developed, ranging from *DLV* ([17]) and *smodels* [22] to SAT-based solvers such as *cmodels* [13] and the conflict-driven solver *clasp* [10]. However, these implementations compute the whole answer set: i.e., *they are not goal-directed* in the fashion of Prolog. Given an answer set program and a query goal `Q`, a goal-directed execution will systematically enumerate—via SLD style call expansions and backtracking—all answer sets that contain the propositions/predicates in `Q`. Other efforts have been made to realize goal-directed implementations (e.g., [5]), however, these approaches can handle only a limited class of programs and/or queries.

In this paper we describe a goal-directed execution method that works for any answer set program as well as for any query. The method relies on *coinductive logic programming* (co-LP) [14]. Co-LP can be regarded as providing an operational semantics, termed co-SLD resolution, for computing greatest fixed points (*gfp*) of logic programs. Co-SLD resolution systematically computes elements of the *gfp* of a program via backtracking [14, 27]. Additionally, calls are allowed to *coinductively succeed* if they unify with one of their ancestor calls [27]. To permit this, each call is stored in the *coinductive hypothesis set* (CHS) as the call is made. A more detailed introduction to co-LP and co-SLD resolution can be found in Appendix A.

A goal-directed method for executing answer set programs is analogous to top-down, SLD style resolution for Prolog, while current popular methods for ASP are analogous to bottom-up methods that have been used for evaluating Prolog (and Datalog) programs [25]. A goal-directed execution method for answering queries for an answer set program has several advantages. The main advantage is that it paves the way to lifting the restriction to finitely groundable programs, and allows realization of ASP with full first-order predicates [20].

In the rest of the paper we develop a goal-directed strategy for executing answer set programs, and prove that it is sound and complete with respect to the method of Gelfond and Lifschitz. We restrict ourselves to only propositional (grounded) answer set programs in this paper; work is in progress to extend our goal-directed method to predicate answer set programs [20, 21]. Note that the design of a top-down goal-directed execution strategy for answer set programs has been regarded as quite a challenging problem [3]. As pointed out in [6], the difficulty in designing a goal-directed method for ASP comes about due to the absence of a *relevance property* in *stable model semantics*, on which answer set

PPDP'12, September 19–21, 2012, Leuven, Belgium.

programming is based [6, 23, 24]. We will introduce a modified relevance property that holds for our goal-directed method and guarantees that partial answer sets computed by our method can be extended to complete answer sets.

2. Answer Set Programming (ASP)

Answer Set Programming (ASP) [12] (A-Prolog [11] or AnsProlog [3]) is a declarative logic programming paradigm which encapsulates non-monotonic or common sense reasoning. The rules in an ASP program are of the form:

`p :-` q_1`, ...,` q_m`, not` r_1`, ..., not` r_n`.`

where $m \geq 0$ and $n \geq 0$. Each of `p` and q_i ($\forall i \leq m$) is a literal, each `not` r_j ($\forall j \leq n$) is a *naf-literal* (`not` is a logical connective called *negation as failure (naf)* or *default negation*). The semantics of an Answer Set program `P` is given via the Gelfond-Lifschitz method [3] in terms of the answer sets of the program `ground(P)`, obtained by grounding the variables in the program `P`.

Gelfond-Lifschitz Transform (GLT) Given a grounded Answer Set program `P` and a candidate answer set `A`, a residual program `R` is obtained by applying the following transformation rules: for all literals `L` $\in$ `A`,

1. Delete all rules in `P` which have `not L` in their body.
2. Delete all the remaining naf-literals (of the form `not M`) from the bodies of the remaining rules.

The least fixed-point (say, `F`) of the residual program `R` is next computed. If `F` $=$ `A`, then `A` is a *stable model* or an *answer set* of `P`.

ASP can also have rules of the form:

`:-` q_1`, ...,` q_m`, not` r_1`, ..., not` r_n`.`

`p :-` q_1`, ...,` q_m`, not` r_1`, ..., not` r_n`, not p.`

These rules capture the non-monotonic aspect of Answer Set Programming. Consider an example rule:

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

Following the Gelfond-Lifschitz method (GL method) outlined above, this rule restricts `q` (and `p`) to not be in the answer set (unless `p` happens to be in the answer set via other rules, in which case due to presence of `not p` this rule will be removed while generating the residual program). Note that even though an answer set program can have other rules to establish that `q` is in the answer set, adding the rule above forces `q` to not be in the answer set unless `p` succeeds through another rule, thus making ASP non-monotonic.

3. Goal-directed ASP Issues

Any normal logic program can also be viewed as an answer set program. However, ASP adds complexity to a normal logic program in two ways. In addition to the standard Prolog rules, it allows:

1. Cyclical rules which when used to expand a call to a subgoal `G` lead to a recursive call to `G` through an even (but non-zero) number of negations. For example, given the program P1 below:

   ```
   p :- not q.      ... Rule P1.a
   q :- not p.      ... Rule P1.b
   ```

   Ordinary logic programming execution for the query `?- p.` (or `?- q.`) will lead to non-termination. However, ASP will produce two answer sets: {`p, not q`} and {`q, not p`}. Expanding the call `p` using Rule P1.a in the style of SLD resolution will lead to a recursive call to `p` that is in scope of two negations (`p` $\rightarrow$ `not q` $\rightarrow$ `not not p`). Such rules are termed *ordinary rules*. Rule P1.b is also an ordinary rule, since if used for expanding the call to `q`, it will lead to a recursive call to `q` through two negations. For simplicity of presentation, all non-cyclical rules will also be classified as ordinary rules.

2. Cyclical rules which when used to expand a call to subgoal `G` lead to a recursive call to `G` that is in the scope of an odd number of negations. Such recursive calls are known as *odd loops over negation* (OLONs). For example, given the program P2 below:

   ```
   p :- q, not p, r. ... Rule P2.a
   ```

   a call to `p` using Rule P2.a will eventually lead to a call to `not p`. Under ordinary logic programming execution, this will lead to non-termination. Under ASP, however, the program consisting of Rule P2.a has {`not p, not q, not r`} as its answer set. For brevity, we refer to rules containing OLONs as *OLON rules*.

Note that a rule can be both an ordinary rule and an OLON rule, since given a subgoal `G`, its expansion can lead to a recursive call to `G` through both even and odd numbers of negations along different expansion paths. For example in program P3 below, Rule P3.a is both an ordinary rule and an OLON rule.

```
p :- q, not r.     ... Rule P3.a
r :- not p.        ... Rule P3.b
q :- t, not p.     ... Rule P3.c
```

Our top-down method requires that we properly identify and handle both ordinary and OLON rules. We will look at each type of rule in turn, followed by the steps taken to ensure that our method remains faithful to the GL method.

3.1 Ordinary Rules

Ordinary rules such as rules P1.a and P1.b in program P1 above exemplify the cyclical reasoning in ASP. The rules in the example force `p` and `q` to be mutually exclusive, i.e., either `p` is true or `q` is true, but not both. One can argue the reasoning presented in such rules is cyclical: If `p` is in the answer set, then `q` cannot be in the answer set, and if `q` is not in the answer set, then `p` must be in the answer set.

Given a goal, `G`, and an answer set program comprised of only ordinary rules, `G` can be executed in a top-down manner using coinduction, through the following steps:

- Record each call in the CHS. The recorded calls constitute the coinductive hypothesis set, which is the potential answer set.
- If at the time of the call, the call is already found in the CHS, it succeeds coinductively and finishes.
- If the current call is not in the CHS, then expand it in the style of ordinary SLD resolution (recording the call in the CHS prior to expansion).
- Simplify `not not p` to `p`, whenever possible, where `p` is a proposition occurring in the program.
- If success is achieved with no goals left to expand, then the coinductive hypothesis set contains the (partial) answer set.

The top-down resolution of query `p` with program P1 will proceed as follows.

`:- p`	CHS = {}
	(expand `p` by Rule P1.a)
`:- not q`	CHS = {`p`}
	(expand `q` by Rule P1.b)

Note that we will list all literals that are true in a given answer set. Conventionally, an answer set is specified by listing only the positive literals that are true; those not listed in the set are assumed to be false.

`:- not not p`	CHS = {`p, not q`}
	(simplify `not not p` $\rightarrow$ `p`)
`:- p`	CHS = {`p, not q`}
	(coinductive success: `p` $\in$ CHS)
`:-` $\square$	success: answer set is {`p, not q`}

Note that the maintenance of the coinductive hypothesis set (CHS) is critical. If a call is encountered that is already in the CHS, it should not be expanded, it should simply (coinductively) succeed. Note that the query `q` will produce the other answer set {`q, not p`} in a symmetrical manner. Note also that the query `not q` will also produce the answer set {`p, not q`} as shown below. Thus, answers to negated queries can also be computed, *if we apply the coinductive hypothesis rule to negated goals also, i.e., a call to `not p` succeeds, if an ancestor call to `not p` is present*:

`:- not q`	CHS = {}
	(expand `q` by Rule P1.b)
`:- not not p`	CHS = {`not q`}
	(`not not p` $\rightarrow$ `p`)
`:- p`	CHS = {`p, not q`}
	(expand `p` by Rule P1.a)
`:- not q`	CHS = {`p, not q`}
	(coinductive success for `not q`)
`:-` $\square$	success: answer set is {`p, not q`}

3.2 OLON Rules

Our goal-directed procedure based on coinduction must also work with OLON rules. OLON rules are problematic because their influence on answer sets is indirect. Under ASP, rules of the form

`p :-` q_1`,` q_2`, ...,` q_k`, not p.`

hold only for those (stable) models in which `p` succeeds through other rules in the program or at least one of the q_i's is false. Note that a headless rule of the form:

`:-` q_1`,` q_2`, ...,` q_n`.`

is another manifestation of an OLON rule, as it is equivalent to the rule:

`p :-` q_1`,` q_2`, ...,` q_n`, not p.`

where `p` is a literal that does not occur anywhere else in the program, in the sense that the stable models for the two rules are identical.

Without loss of generality, consider the simpler rule:

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

For an interpretation to be a (stable) model for this rule, either `p` must succeed through other rules in the program or `q` must be false. Two interesting cases arise: (i). `p` is true through other rules in the program. (ii) `q` is true through other rules in the program.

For case (i), if `p` is true through other means in the program, then according to the Gelfond-Lifschitz method, it is in the answer set, and the OLON rule is taken out of consideration due to the occurrence of `not p` in its body. For case (ii), if `q` is true through other means and the rule is still in consideration due to `p` not being true through other rules in the program, then there are no answer sets, as `q` is both true and false. Thus, the answer set of the program P4 below is: {`p, not q`}.

`p :- q, not p.`	... Rule P4.1
`p.`	... Rule P4.2

while there is no answer set for program P5 below:

`p :- q, not p.`	... Rule P5.1
`q.`	... Rule P5.2

Given an OLON rule with `p` as its head and the query `p`, execution based on co-SLD resolution will fail, *if we require that the coinductive hypothesis set (CHS) remains consistent at all times.* That is, if we encounter the goal `g` (resp. `not g`) during execution and `not g` $\in$ CHS (resp. `g` $\in$ CHS), then the computation fails and backtracking ensues.

As another example, consider the program containing rule P4.1 (which has `p` in its head), but not rule P4.2, and the query `:- p`. When execution starts, `p` will be added to the CHS and then expanded by rule P4.1; if the call to `q` fails, then the goal `p` also fails. Alternatively, if `q` succeeds due to other rules in the program, then upon arriving at the call `not p`, failure will ensue, since `not p` is inconsistent with the current CHS (which equals {`p`, `q`} prior to the call `not p`).

Thus, OLON rules do not pose any problems in top-down execution based on coinduction, however, given an OLON rule with `p` as its head, if `p` can be inferred by other means (i.e., through ordinary rules) then the query `p` should succeed. Likewise, if `q` succeeds by other means and `p` does not, then we should report a failure (rather, report the absence of an answer set; note that given our conventions, CHS = {} denotes no answer set). We discuss how top-down execution of OLON rules is handled in Section 3.4.

3.3 Coinductive Success Under ASP

While our technique's use of co-SLD resolution has been outlined above, it requires some additional modification to be faithful to the Gelfond-Lifschitz method. Using normal coinductive success, our method will compute the *gfp* of the residual program after the GL transform, while the GL method computes the *lfp*. Consider Program P6 below:

`p :- q.`	... Rule P6.1
`q :- p.`	... Rule P6.2

Our method based on coinduction will succeed for queries `:- p` and `:- q` producing the answer set {`p, q`} while under the GL method, the answer set for this program is {`not p, not q`}. Our top-down method based on coinduction really computes the *gfp* of the original program. The GL method computes a fixed point of the original program (via the GL transformation and then computation of the *lfp* of the residual program) that is in between the *gfp* and the *lfp* of the original program. In the GL method, *direct cyclical reasoning is not allowed, however, cyclical reasoning that goes through at least one negated literal is allowed*. Thus, under the GL method, the answer set of program P6 does not contain a single positive literal, while there are two answer sets for the program P1 given earlier, each with exactly one positive literal, even though both programs P1 and P6 have only cyclical rules.

Our top-down method can be modified so that it produces answer sets consistent with the GL method: *a coinductive recursive call can succeed only if it is in the scope of at least one negation.* In other words, the path from a successful coinductive call to its ancestor call must include a call to `not`.

This restriction disallows inferring `p` from rules such as

```
p :- p.
```

With this operational restriction in place, the CHS will never contain a positive literal that is in the *gfp* of the residual program obtained after the GLT, but not in its *lfp*. To show this, let us assume that, for some ASP program, a call to `p` will always encounter at least one recursive call to `p` with no intervening negation. In such a case, `p` will never be part of any answer set:

- Under our goal-directed method, any call to `p` will fail when a recursive call is encountered with no intervening negation.
- Under the GL method, `p` will never be in the *lfp* of the residual. Even if a rule for `p` is present in the residual and all other dependencies are satisfied, the rule will still depend on the recursive call to `p`.

3.4 NMR Consistency Check

To summarize, the workings of our goal-directed strategy are as follows: given a goal G, perform co-SLD resolution while restricting coinductive success as outlined in Section 3.3. The CHS serves as the potential answer set. A successful answer will be computed only through ordinary rules, as all OLON rules will lead to failure due to the fact that `not h` will be encountered with proposition h present in the CHS while expanding with an OLON rule whose head is h. Once success is achieved, the answer set is the CHS. As discussed later, this answer set may be partial.

The answer set produced by the process above is only a potential answer set. Once a candidate answer set has been generated by co-SLD resolution as outlined above, the set has to be checked to see that it will not be rejected by an OLON rule. Suppose there are n OLON rules in the program of the form:

q_i `:-` B_i`.`

where $1 \leq i \leq n$ and each B_i is a conjunction of goals. Each B_i must contain a direct or indirect call to the respective q_i which is in the scope of odd number of negations in order for q_i `:-` B_i`.` to qualify as an OLON rule.

If a candidate answer set contains q_j $(1 \leq j \leq n)$, then each OLON rule whose head matches q_j must be taken out of consideration (this is because B_j leads to `not(`q_j`)` which will be false for this candidate answer set). For all the other OLON rules whose head proposition q_k $(1 \leq j \leq n)$ is not in the candidate answer set, their bodies must evaluate to false w.r.t. the candidate answer set, i.e., for each such rule, B_k must evaluate to false w.r.t. the candidate answer set.

The above restrictions can be restated as follows: a candidate answer set must satisfy the formula $q_i \vee$ `not` B_i $(1 \leq i \leq n)$ for each OLON rule q_i `:-` B_i`.` $(1 \leq i \leq n)$ in order to be reported as the final answer set. Thus, for each OLON rule, the check

chk_q_i `:-` q_i`.`
chk_q_i `:- not` B_i`.`

is constructed by our method. Furthermore, `not` B_i will be expanded to produce a chk_q_i clause for each literal in B_i. For example, if B_i represented the conjunction of literals `s, not r, t` in the above example, the check created would be:

chk_q_i `:-` q_i`.`
chk_q_i `:- not s.`
chk_q_i `:- r.`
chk_q_i `:- not t.`

A candidate answer set must satisfy each of these checks in order to be reported as a solution. This is enforced by rolling the checks into a single call, termed `nmr_chk`:

`nmr_chk :-` chk_q_1`,` chk_q_2`, ...`chk_q_n`.`

Now each query Q is transformed to `Q, nmr_chk.` before it is posed to our goal-directed system. One can think of Q as the generator of candidate answer sets and `nmr_chk` as the filter. If `nmr_chk` fails, then backtracking will take place and Q will produce another candidate answer set, and so on. Backtracking can also take place within Q itself when a call to p (resp. `not p`) is encountered and `not p` (resp. p) is present in the CHS. Note that the CHS must be a part of the execution state, and be restored upon backtracking.

4. Goal-directed Execution of Answer Set Programs

We next describe our general goal-directed procedure for computing answer sets.

4.1 Dual Rules

For simplicity, we add one more step to the process. Similarly to Alferes et al [1], for each rule in the program, we introduce its dual. That is, given a proposition H's definition (B_i's are conjunction of literals):

H `:-` B_1`.`
H `:-` B_2`.`
...
H `:-` B_n`.`

we add the dual rule

`not H :- not` B_1`, not` B_2`, ..., not` B_n`.`

If a proposition q appears in the body of a rule but not in any of the rule heads, then the fact

`not q.`

is added. Note that adding the dual rules is not necessary; it only makes the exposition of our goal-directed method easier to present and understand.

4.2 Goal-directed Method for Computing Answer Sets

Given a propositional query `:-` Q and a propositional answer set program P, the goal-directed procedure works as described below. Note that the execution state is a pair (G, S), where G is the current goal list, and S the current CHS.

1. Identify the set of ordinary rules and OLON rules in the program.
2. Assert a chk_q_i rule for every OLON rule with q_i as its head and build the `nmr_check` as described in Section 3.4.
3. For each ordinary rule and chk_q_i rule, construct its dual version.
4. Append the `nmr_check` to the query.
5. Set the initial execution state to: (`:-` G_1`, ...,` G_n, {}).
6. Non-deterministically reduce the execution state using the following rules:

   (a) *Call Expansion:*
   (`:-` G_1`, ..,` G_i`, ..,` G_n, S)
   $\rightarrow$ (`:-` G_1`, ..,` B_1`, ..,` B_m`, ..,` G_n, S $\cup \{G_i\}$)
   where G_i matches the rule G_i `:-` B_1`, ...,` B_m`.` in P, $G_i \notin$ S and `not` $G_i \notin$ S.

   (b) *Coinductive Success:*
   (`:-` G_1`, ..,` G_{i-1}`,` G_i`,` G_{i+1}`, ..,` G_n, S)
   $\rightarrow$ (`:-` G_1`, ..,` G_{i-1}`,` G_{i+1}`, ..,` G_n, S)
   if $G_i \in$ S and either:
   i. G_i is not a recursive call or
   ii. G_i is a recursive call in the scope of a non-zero number of intervening negations.

   (c) *Inductive Success:*
   (`:-` G_1`, ..,` G_{i-1}`,` G_i`,` G_{i+1}`, ..,` G_n, S)
   $\rightarrow$ (`:-` G_1`, ..,` G_{i-1}`,` G_{i+1}`, ..,` G_n, S $\cup \{G_i\}$)
   if G_i matches a fact.

   (d) *Coinductive Failure:*
   (`:-` G_1`, ..,` G_i`, ..,` G_n, S) $\rightarrow$ (fail, S)
   if either:
   i. `not` $G_i \in$ S or
   ii. $G_i \in$ S and G_i is a recursive call without any intervening negations.

   (e) *Inductive Failure:*
   (`:-` G_1`, ..,` G_i`, ..,` G_n, S) $\rightarrow$ (fail, S)
   if G_i has no matching rule in P.

   (f) *Print Answer:*
   (`:- true`, S) $\rightarrow$ success: S is the answer set
   where '`:- true`' $\equiv$ empty goal list

Note that when all the goals in the query are exhausted, execution of `nmr_chk` begins. Upon failure, backtracking ensues, the state is restored and another rule tried. Note that negated calls are expanded using dual rules as in [1], so it is not necessary to check whether the number of intervening negations between a recursive call and its ancestor is even or odd. (See the call expansion rule above). A detailed example of goal-directed execution can be found in Appendix B. Next we discuss a few important issues:

Identifying OLON and Ordinary Rules Given a propositional answer set program `P`, OLON rules and ordinary rules can be identified by constructing and traversing the call graph. The complexity of this traversal is $O(|\texttt{P}| * n)$, where n is the number of propositional symbols occurring in the head of clauses in `P` and $|\texttt{P}|$ is a measure of the program size. Note also that during the execution of a query `Q`, we need not make a distinction between ordinary and OLON rules; knowledge of OLON rules is needed only for creating the `nmr_chk`.

Partial Answer Set Our top-down procedure might not generate the entire answer set. It may generate only the part of the answer set that is needed to evaluate the query. Consider program P7:

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

Under goal-directed execution, the query `:- q.` for program P7 will produce only {`q, not p`} as the answer since the rules defining `r` and `s` are completely independent of rules for `p` and `q`. One could argue that this is an advantage of a goal-directed execution strategy rather than a disadvantage, as only the relevant part of the program will be explored. In contrast, if the query is `:- q, s`, then the right answer {`q, not p, s, not r`} will be produced by the goal-directed execution method. Thus, the part of the answer set that gets computed depends on the query. Correct maintenance of the CHS throughout the execution is important as it ensures that only consistent and correct answer sets are produced.

5. Soundness and Correctness of the Goal-directed Method

We will now show the correctness of our goal-directed execution method by showing it to be sound and complete with respect to the GL method. First, we will examine the modified relevance property which holds for our method.

5.1 Relevance

As we stated in the introduction, one of the primary problems with developing a goal-directed ASP implementation is the lack of a relevance property in stable model semantics. Dix introduces relevance by stating that, "given any semantics SEM and a program P, it is perfectly reasonable that the truth-value of a literal L, with respect to SEM(P), only depends on the subprogram formed from the relevant rules of P with respect to L" [6]. He formalizes this using the dependency-graph of P, first establishing that

- "$dependencies\_of(X) := \{\text{A : X depends on A}\}$, and
- rel_rul(P,X) is the set of relevant rules of P with respect to X, i.e. the set of rules that contain an $A \in dependencies\_of(X)$ in their heads" [6]

and noting that the dependencies and relevant rules of $\neg X$ are the same as those of X [6]. He then defines relevance as, for all literals L:

$$SEM(P)(L) = SEM(rel\_rul(P, L))(L) \tag{1}$$

The relevance property is desirable because it would ensure that a partial answer set computed using only relevant rules for each literal could be extended into a complete answer set. However, stable model semantics do not satisfy the definition as given. This is because OLON rules can alter the meaning of a program and the truth values of individual literals without occurring in the set of relevant rules [6, 23]. For instance, an irrelevant rule of the form `p :- not p.` when added to an answer set program P, where P has one or more stable models and `p` does not occur in P, results in a program that has no stable models.

Approaches such as [23] have addressed the lack of a relevance property by modifying stable model semantics to restore relevance. However, our implementation can be viewed as restoring relevance by expanding the definition of relevant rules to include all OLON rules in a program. Because the NMR check processes every OLON rule, it has the effect of making the truth value of every literal in a program dependent on such rules. That is,

$$\begin{aligned} nmr\_rel\_rul(P, L) &= rel\_rul(P, L) \cup O, \\ O &= \{\text{ R: R is an OLON rule in P }\} \end{aligned} \tag{2}$$

Using nmr_rel_rul(P,L) in place of rel_rul(P,L), a modified version of equation 1 above holds for our semantics:

$$SEM(P)(L) = SEM(nmr\_rel\_rule(P, L))(L) \tag{3}$$

As a result, any partial model returned by our semantics is guaranteed to be a subset of one or more complete models.

5.2 Soundness

Theorem 1. *For the non-empty set X returned by successful top-down execution of some program P, the set of positive literals in X will be an answer set of R, the set of rules of P used during top-down execution.*

Proof. Let us assume that top-down execution of a program P has succeeded for some query Q consisting of a set of literals in P, returning a non-empty set of literals X. We can observe that $R \subseteq \bigcup_{L \in Q} nmr\_rel\_rul(P, L)$: for each positive literal in Q, one rule with the literal in its head will need to succeed, for each negative literal in Q all rules with the the positive form of the literal in their head will need to fail, and the resulting set must satisfy the NMR check. We will show that X is a valid answer set of R using the GL method. First, because X may contain negative literals and the residual program produced by the GL method is a positive one, let us remove any rules in R containing the positive version of such literals as a goal, and then remove the negated literals from X to obtain X'. Because our algorithm allows negative literals to succeed if and only if all rules for the positive form fail or no such rules exist, only rules which failed during execution will be removed by this step. Next, let us apply the GL transformation using X' as the candidate answer set to obtain the residual program R'. This will remove rules containing the negation of any literal in X' and remove any negated goals from the remaining rules.

We know that X' will be an answer set of R if and only if X' = LFP(R'). Now let us examine the properties of R'. As positive literals, we know that each literal in X' must occur as the head of a rule in R which succeeded during execution. Because such rules would have failed if the negation of any goal was present in the CHS, we know that such rules would not have been eliminated from the residual program by the GL transformation, and are thus still present in R' save for the removal of any negated goals. Because any rules containing the negation of a literal in X had to fail during execution, at least one goal in each of these rules must have failed, resulting in the negation of the goal being added to the CHS. Furthermore, because the NMR check applies the negation of each

OLON rule, again the negation of some goal in each such rule must have been added to the CHS. Thus any rule which failed during execution and yet was included in R will have been removed from R'. Finally, because our algorithm allows coinductive success to occur only in the scope of at least one negation, the removal of negated goals from the residual program will ensure that R' contains no loops. Because the remaining rules in R' must have succeeded during execution, their goals must have been added to the CHS, and therefore those goals consisting of positive literals form X'. Thus R' is a positive program with no loops, and each literal in X' must appear as the head of some rule in R' which is either a fact or whose goals consist only of other elements in X'. Therefore the least fixed point of R' must be equal to X', and X' must be an answer set of R. □

Theorem 2. *Our top-down execution algorithm is sound with respect to the GL method. That is, for the non-empty set of literals X returned by successful execution of some program P, the set of positive literals in X is a subset of one or more answer sets of P.*

Proof. As shown above, the positive literals in the set returned by successful execution of P will be an answer set of $R \subseteq \bigcup_{L \in Q} nmr\_rel\_rul(P, L)$. Because R will always contain all OLON rules in P, no unused rules in P are capable of affecting the truth values of the literals in X. Thus the modified definition of relevance holds for all literals in X under our semantics and the partial answer set returned by our algorithm is guaranteed to be extensible to a complete one. Thus our algorithm for top-down execution is sound with respect to the GL method. □

5.3 Completeness

Theorem 3. *Our top-down execution algorithm is complete with respect to the GL method. That is, for a program P, any answer set valid under the GL method will succeed if used as a query for top-down execution. In addition, the set returned by successful execution will contain no additional positive literals.*

Proof. Let X be a valid answer set of P obtained via the GL method. Then there exists a resultant program P' obtained by removing those rules in P containing the negation of any literal in X and removing any additional negated literals from the goals of the remaining rules. Furthermore, because X is a valid answer set of P, X = LFP(P'). This tells us that for every literal $L \in X$ there is a rule in P' with L as its head, which is either a fact or whose goals consist only of other literals in X.

Let us assume that X is posed as a query for top-down execution of P. As we know that each $L \in X$ has a rule in P' with L as its head and whose positive goals are other literals in X, we know that such a rule also exists in P, with the possible addition of negated literals as goals. However, we know that these negated literals must succeed, that is, all rules with the positive form of such literals in their heads must fail, either by calling the negation of some literal in the answer set or by calling their heads recursively without an intervening negation. Were this not the case, these rules would remain in P', their heads would be included in LFP(P') and X would not be a valid answer set of P. Therefore, a combination of rules may be found such that each literal in X appears as the head of at least one rule which will succeed under top-down execution, and whose positive goals are all other literals in X. Furthermore, because each literal in the query must be satisfied and added to the CHS, and any rule with a goal whose negation is present in the CHS will fail, such a combination of rules will eventually be executed by our algorithm. Because such rules would also be present in P', we know that they cannot add additional positive literals to the CHS, as these would be part of LFP(P'), again rendering X invalid.

Table 1. N-Queens Problem; 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

Table 2. MxN-Pigeons Problem (No Solution for M>N)

Problem	*Galliwasp*	*clasp*	*cmodels*	*smodels*
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

Table 3. MxN-Coloring problem (No Solution for M=3)

Problem	*Galliwasp*	*clasp*	*cmodels*	*smodels*
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

This leaves the NMR check, which ensures the set returned by our algorithm satisfies all OLON rules in P. However, we know this is the case, as the subset of positive literals in the CHS is equal to X. Because X is a valid answer set of P, there cannot be any rule in P which renders X invalid, and thus the NMR check must be satisfiable by a set of literals containing X. We also know that the NMR check will not add additional positive literals to the CHS, as any rules able to succeed would be present in P' and thus present in LFP(P').

Therefore any valid answer set X of a program P must succeed if posed as a query for top-down execution of P. Thus our top-down algorithm is complete with respect to the GL method. □

6. Performance Results

The goal-directed method described in this paper has been implemented in our system *Galliwasp*. In addition to the goal-directed method presented here, *Galliwasp* incorporates various other techniques to improve performance, including incremental enforcement of the NMR check [19]. Tables 1, 2 and 3 give performance results for some example programs. For the purpose of comparison, results for *clasp*, *cmodels* and *smodels* are also given.

The *Galliwasp* system consists of two programs, a compiler and an interpreter. The times given are for our interpreter using

a compiled program and for the other solvers reading a program grounded by lparse. Neither compilation nor grounding times are factored into the results. A timeout of 600 seconds was enforced, with the instances which timed out listed as N/A in the tables.

As these results demonstrate, our goal-directed method is practical and can be efficiently implemented. While additional performance increases are possible, the *Galliwasp* interpreter is already significantly faster than *smodels* in almost every case and comparable to *clasp* and *cmodels* in most cases.

7. Discussion and Related Work

There are many advantages of top-down goal-directed execution of answer set programs, the main one being that it paves the way to answer set programming with predicates. The first step is to extend our method to *datalog answer set programs*, i.e., programs that allow only constants and variables as arguments in the predicates they contain [20].

Another advantage of goal-directed execution is that answer set programming can be made to work more naturally with other extensions that have been developed within logic programming, such as constraint programming, abduction, parallelism, probabilistic reasoning, etc. This leads to more sophisticated applications. Timed planning, i.e., planning in the presence of real-time constraints, is one such example [2].

With respect to related work, a top-down, goal-directed execution strategy for ASP has been the aim of many researchers in the past. Descriptions of some of these efforts can be found in [1, 4, 5, 8, 9, 15, 23, 24, 26]. The strategy presented in this paper is based on one presented by several of this paper's authors in previous work [14, 21]. However, the strategy presented in those works was limited to call-consistent or order-consistent programs. While the possibility of expansion to arbitrary ASP programs was mentioned, it was not expanded upon, and the proofs of soundness and completeness covered only the restricted cases [21].

A query-driven procedure for computing answer sets via an abductive proof procedure has been explored [7, 16]: a consistency check via integrity constraints is done before a negated literal is added to the answer set. However, "this procedure is not always sound with respect to the above abductive semantics of NAF" [16]. Alferes et al [1] have worked in a similar direction, though this is done in the context of abduction and again goal-directedness of ASP is not the main focus. Gebser and Schaub have developed a tableau based method which can be regarded as a step in this direction, however, the motivation for their work is completely different [9].

Bonatti, Pontelli and Tran [5] have proposed credulous resolution, an extension of earlier work of Bonatti [4], that extends SLD resolution for ASP. However, they place restrictions on the type of programs allowed and the type of queries allowed. Their method can be regarded as allowing coinductive success to be inferred only for negated goals. Thus, given query `:- p` and program P1, the execution will look as follows: `p` $\rightarrow$ `not q` $\rightarrow$ `not not p` $\rightarrow$ `not q` $\rightarrow$ `success`. Compared to our method, their method performs extra work. For example, if rule P1.1 is changed to `p :- big_goal, not q`, then `big_goal` will be executed twice. The main problem in their method is that since it does not take coinduction for positive goals into account, knowing when to succeed inductively and when to succeed coinductively is undecidable. For this reason, their method works correctly only for a limited class of answer set programs (for example, answers to negated queries such `?-not p` cannot be computed in a top-down manner). In contrast, our goal-directed method works correctly for all types of answer set programs and all types of queries.

Pereira's group has done significant work on defining semantics for normal logic programs and implementing them, including implementation in a top-down fashion [1, 23, 24]. However, their approach is to modify stable model semantics so that the property of relevance is restored [23]. For this modified semantics, goal-directed procedures have been designed [24]. In contrast, our goal is to stay faithful to stable model semantics and answer set programming.

8. Conclusions

The main contribution of our paper is to present a practical, top-down method for goal-directed execution of Answer Set programs along with proofs of soundness and completeness. Our method stays faithful to ASP, and works for arbitrary answer set programs as well as arbitrary queries. Other methods in the literature either change the semantics, or work for only restricted programs or queries. Our method achieves this by relying on the coinductive logic programming paradigm. Details of our method were presented, along with proofs of soundness and correctness, and some preliminary performance results. A goal-directed procedure has many advantages, the main one being that execution of answer set programs does not have to be restricted to only finitely groundable ones. Our work thus paves the way for developing execution procedures for ASP over predicates. A goal-directed strategy permits an easier integration with other extensions of logic programming, which in turn makes it possible to develop more interesting applications of ASP and non-monotonic reasoning. Our current work is focused on refining our implementation to improve efficiency and add support for features such as constraints and predicates.

9. Acknowledgments

Thanks to Michael Gelfond, Vladimir Lifschitz, Enrico Pontelli and Feliks Kluźniak for discussions and feedback.

References

[1] J. J. Alferes, L. M. Pereira, and T. Swift. 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] A. Bansal. *Towards Next Generation Logic Programming Systems*. PhD thesis, University of Texas at Dallas, 2007.

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

[4] P. Bonatti. Resolution for Skeptical Stable Model Semantics. *Journal of Automated Reasoning*, 27:391–421, November 2001.

[5] P. A. Bonatti, E. Pontelli, and T. C. Son. Credulous Resolution for Answer Set Programming. In *Proceedings of the 23rd national conference on Artificial Intelligence - Volume 1*, AAAI'08, pages 418–423. AAAI Press, 2008.

[6] J. Dix. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, 22:257–288, 1995.

[7] K. Eshghi and R. A. Kowalski. Abduction Compared with Negation by Failure. In *ICLP*, ICLP'89, pages 234–254, 1989.

[8] J. Fernández and J. Lobo. A Proof Procedure for Stable Theories. In *CS-TR-3034*, Computer Science Technical Report Series. University of Maryland, 1993.

[9] M. Gebser and T. Schaub. Tableau Calculi for Answer Set Programming. In *Proceedings of the 22nd international conference on Logic Programming*, ICLP'06, pages 11–25. Springer-Verlag, 2006.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Clasp: A Conflict-Driven Answer Set Solver. In *Proceedings of the 9th international conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pages 260–265. Springer-Verlag, 2007.

[11] M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 413–451. Springer-Verlag, 2002.

[12] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth international conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[13] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In *Proceedings of the 19th national conference on Artifical Intelligence*, AAAI'04, pages 61–66. AAAI Press, 2004.

[14] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive Logic Programming and Its Applications. In *Proceedings of the 23rd international conference on Logic Programming*, ICLP'07, pages 27–44. Springer-Verlag, 2007.

[15] A. Kakas and F. Toni. Computing Argumentation in Logic Programming. *Journal of Logic and Computation*, 9(4):515–562, 1999.

[16] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[17] N. Leone, G. Pfeifer, and W. Faber. DLV. http://www.dbai.tuwien.ac.at/proj/dlv.

[18] J. Lloyd. *Foundations of Logic Programming*. Symbolic Computation: Artificial Intelligence. Springer-Verlag, 1987.

[19] K. Marple and G. Gupta. Galliwasp: A Goal-Directed Answer Set Solver. In *Proceedings of the 22nd international symposium on Logic-based Program Synthesis and Transformation*, LOPSTR '12, pages 85–99. Katholieke Universiteit Leuven, 2012.

[20] R. Min. *Predicate Answer Set Programming with Coinduction*. PhD thesis, University of Texas at Dallas, 2010.

[21] R. Min, A. Bansal, and G. Gupta. Towards Predicate Answer Set Programming via Coinductive Logic Programming. In *AIAI*, pages 499–508. Springer, 2009.

[22] I. Niemelä and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer-Verlag, 1997.

[23] L. Pereira and A. Pinto. Revised Stable Models - A Semantics for Logic Programs. In *Progress in Artificial Intelligence*, volume 3808 of *Lecture Notes in Computer Science*, pages 29–42. Springer-Verlag, 2005.

[24] L. Pereira and A. Pinto. Layered Models Top-Down Querying of Normal Logic Programs. In *Practical Aspects of Declarative Languages*, volume 5418 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 2009.

[25] K. Sagonas, T. Swift, and D. Warren. XSB as an Efficient Deductive Database Engine. *ACM SIGMOD Record*, 23(2):442–453, 1994.

[26] Y. Shen, J. You, and L. Yuan. Enhancing Global SLS-Resolution with Loop Cutting and Tabling Mechanisms. *Theoretical Computer Science*, 328(3):271–287, 2004.

[27] L. Simon. *Extending Logic Programming with Coinduction*. PhD thesis, University of Texas at Dallas, 2006.

A. Co-SLD Resolution

As mentioned in the introduction, our goal-directed method relies on *coinductive logic programming* (co-LP) [14]. Co-SLD resolution, the operational semantics of coinduction, is briefly described below. The semantics is limited to *regular proofs*, i.e., those cases where the infinite behavior is obtained by infinite repetition of a finite number of finite behaviors.

Consider the logic programming definition of a stream (list) of numbers as in program R1 below:

```
stream([]).
stream([H|T]) :- number(H), stream(T).
```

Under SLD resolution, the query `?- stream(X)` will systematically produce all finite streams one by one starting from the `[]` stream. Suppose now we remove the base case and obtain the program R2:

```
stream([H|T]) :- number(H), stream(T).
```

In the program R2, the meaning of the query `?- stream(X)` is semantically null under standard logic programming. In the co-LP paradigm the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand (or co-Herbrand) universe, infinitary Herbrand (or co-Herbrand) base [18], and maximal models (computed using greatest fixed-points)* [27]. The operational semantics under coinduction is as follows [27]: a predicate call $\texttt{p}(\bar{t})$ succeeds if it unifies with one of its ancestor calls. Thus, every time a call is made, it has to be remembered. This set of ancestor calls constitutes the *coinductive hypothesis set* (CHS). Under co-LP, infinite *rational* answers can be computed, and infinite rational terms are allowed as arguments of predicates. Infinite terms are represented as solutions to unification equations and the occurs check is omitted during the unification process: for example, `X = [1 | X]` represents the binding of `X` to an infinite list of `1`'s. Thus, in co-SLD resolution, given a single clause

```
p([ 1 | X ]) :- p(X).
```

The query `?- p(A)` will succeed in two resolution steps with the answer `A = [1 | A]`, which is a finite representation of the infinite answer `A = [1, 1, 1, ....]`. Under coinductive interpretation of R2, the query `?- stream(X)` produces all infinite sized streams as answers, e.g., `X = [1 | X]`, `X = [1, 2 | X ]`, etc. Thus, the semantics of R2 is not null, but proofs may be of infinite length. If we take a coinductive interpretation of program R1, then we get all finite and infinite streams as answers to the query `?- stream(X)`.

B. Detailed Execution Example

We now present a larger, more complex example of execution using our goal-directed method. Consider program A1:

```
p :- not q.        ... Rule A1.1
q :- not r.        ... Rule A1.2
r :- not p.        ... Rule A1.3
q :- not p.        ... Rule A1.4
```

Rules A1.1, A1.2 and A1.3 are OLON rules, as calls to propositions `p`, `q`, and `r` in the heads of these rules lead to recursive calls to `p`, `q` and `r` respectively that are in the scope of odd numbers of negations. A1.1 is also an ordinary rule, since in conjunction with rule A1.4, a call to `p` resolved via rule A1.1 will lead to a call to `p` in rule A1.4 that is in the scope of an even number of negations. Thus, the `nmr_check` rule can be defined as:

```
nmr_check :- not chk_p, not chk_q, not chk_r.
chk_p :- not p, not q.
chk_q :- not q, not r.
chk_r :- not r, not p.
```

The duals of the above rules are as follows:

```
not p :- q.          ... Rule A1.6
not q :- r, p.       ... Rule A1.7
not r :- p.          ... Rule A1.8
not chk_p :- p; q.   ... Rule A1.9
not chk_q :- q; r.   ... Rule A1.10
not chk_r :- r; p.   ... Rule A1.11
```

Negated calls are resolved using these dual rules. Now the query `q` will be extended to `q, nmr_chk` and executed as follows:

```
:- q, nmr_chk.      CHS = {}; Rule A1.2
:- not r, nmr_chk.  CHS = {q}; Rule A1.8
:- p, nmr_chk.      CHS = {q, not r}; Rule A1.1
:- not q, nmr_chk.  CHS = {q, not r}
                    fail: backtrack to step 1
:- q, nmr_chk.      CHS = {}; Rule A1.4
:- not p, nmr_chk.  CHS = {q, not p}; Rule A1.6
```

`:- q, nmr_chk.`	CHS = {q, not p}
	coinductive success
`:- nmr_chk.`	CHS = {q, not p}
	execution of q finished
`:- not chk_p, not chk_q, not chk_r.`	
	CHS = {q, not p}
	nmr_chk rule
`:- (p ; q), not chk_q, not chk_r.`	
	CHS = {q, not p}
	not p is in CHS
`:- q, not chk_q, not chk_r.`	
	CHS = {q, not p}
	coinductive success for q
`:- not chk_q, not chk_r.`	
	CHS = {q, not p}; Rule A1.10
`:- (q ; r), not chk_r.`	
	CHS = {q, not p}
	coinductive success for q
`:- not chk_r.`	CHS = {q, not p}; Rule A1.11
`:- r ; p.`	CHS = {q, not p, r}; Rule A1.3
`:- not p ; p.`	CHS = {q, not p, r}
	coinductive success for not p
`:-` □.	success.
	answer set is {`q, not p, r`}