arXiv:1709.00501v1 [cs.LO] 1 Sep 2017

Computing Stable Models of Normal Logic Programs Without Grounding

KYLE MARPLE, ELMER SALAZAR and GOPAL GUPTA∗

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

Abstract

We present a method for computing stable models of normal logic programs, i.e., logic programs extended with negation, in the presence of predicates with arbitrary terms. Such programs need not have a finite grounding, so traditional methods do not apply. Our method relies on the use of a non-Herbrand universe, as well as coinduction, constructive negation and a number of other novel techniques. Using our method, a normal logic program with predicates can be executed directly under the stable model semantics without requiring it to be grounded either before or during execution and without requiring that its variables range over a finite domain. As a result, our method is quite general and supports the use of terms as arguments, including lists and complex data structures. A prototype implementation and non-trivial applications have been developed to demonstrate the feasibility of our method.

KEYWORDS: stable models, predicate, goal-directed

1 Introduction

The addition of negation to logic programming has been the subject of significant research over the last several decades. Both classical negation (where negative inference must be explicitly established) and *negation as failure* (where `not p` is inferred if we fail to establish `p`) have been considered. Of these, negation as failure (NAF) has more interesting applications and has been widely researched.

Including NAF poses the following problem: while a logic program with no negation has a unique minimal model, logic programs with NAF can lead to multiple, incompatible models. Different semantics of negation arise depending on which of these models are deemed acceptable. These semantics include Fitting's semantics, the well-founded semantics of Ross, Gelder and Schlipf, the stable model semantics and a multitude of others (Fitting 1985; Van Gelder et al. 1991; Gelfond and Lifschitz 1988; Ullman 1994). Of the various semantics which have been developed, the stable model semantics is widely regarded as the most expressive (Baral 2003).

However, given the current computation methods, the stable model semantics,

and the answer set programming paradigm inspired by them, are computable only for programs which are finitely groundable. Thus, current methods compute the stable models of a program by first *grounding* the program, i.e., instantiating each program variable with each of the values from its respective domain to derive ground clauses. The stable models are then computed using the grounded program. In most modern implementations, the ground program is suitably transformed and fed to a SAT solver. The models produced by the SAT solver will be the stable models of the original program (Lin and Zhao 2004; Gebser et al. 2007).

There are several problems with the grounding-based approach, the most significant being that only certain classes of programs are guaranteed to have a finite grounding. A logic program with even a single unary term will have an infinite number of groundings due to the fact that, given a single unary function symbol f/1 and a single constant value `a`, the domain over which variables can range is infinite, consisting of {`a, f(a), f(f(a)), f(f(f(a))), ...`}, resulting in infinite number of grounded clauses. For such a program to have a finite grounding, each variable must be restricted to a finite domain. However, even with finite domains, the grounding of a program may be exponentially larger than the original. Secondly, SAT-based and similar approaches compute the complete model of the grounded program. In reality, when solving practical problems, we are often only interested in part of the model. Finally, adding negation can lead to (parts of) the program becoming inconsistent. When this occurs, bottom-up methods that work with grounded programs will declare the whole program to be inconsistent (i.e., no model exists). In practice, it may be desirable to compute answers as long as these answers do not involve the inconsistent part of the program (Marple and Gupta 2014).

The last two problems described above can be resolved by designing goal-directed or query-driven execution methods for computing stable models. Such a method does not use a SAT solver, but rather, given a query, computes a partial stable model containing the query, if one exists. The computation is done in a manner very similar to SLD resolution (Lloyd 1987) for logic programs (Marple et al. 2012; Marple and Gupta 2014). We have presented such a method previously, however, that method only works for propositional (grounded) programs(Marple et al. 2012; Marple and Gupta 2014). In this paper, we build on our previous work to remove the need for grounding. Thus, we develop a query-driven method that can apply the stable model semantics to a normal logic program containing arbitrary terms as well as negation. This is accomplished without grounding the program, either before or during execution. It should be noted that, until recently, such query-driven procedures were considered impossible to develop even for propositional logic programs (Baral and Gelfond 1994).

The key insight in our work is that we use a non-Herbrand universe (in fact, an infinite *superset* of the Herbrand universe) which allows us to guarantee properties required for the correctness of our method while still obtaining useful results. Additionally, coinductive logic programming is used to establish the consistency of mutually dependent (co-)recursive calls (Gupta et al. 2007). *Dual rules* are used both to simplify the handling of negation and to provide constructive negation.

It should be noted that top-down, goal-directed implementations which support

predicates have been designed for the well-founded semantics by extending Prolog systems, for example, with *tabling* (Chen et al. 1995). However, the well-founded semantics can be too weak for many applications, as it declares the truth value of many interesting atoms to be unknown. The stable model semantics is more expressive, but, to date, there has been no satisfactory solution to the problem of computing stable models of arbitrary predicate logic programs. Those solutions that have been proposed either greatly restrict the types of programs that can be handled or ground the program incrementally during execution (Dal Palù et al. 2009; Dao-Tran et al. 2012; Lefvre and Nicolas 2009a; Lefvre and Nicolas 2009b). Thus, our research makes important contributions:

- It presents a top-down, query-driven method that can execute normal logic programs with arbitrary predicates, thus solving a problem that was hitherto considered unsolvable.

- Our method can be thought of as providing an operational semantics to normal logic programs with predicates (or, Prolog with negation) under the stable model semantics. This can be combined with other advanced features of logic programming such as constraints (Jaffar and Lassez 1987) to develop extremely powerful applications in an elegant manner, such as automated planning under real-time constraints (Bansal et al. 2010).

- The stable model semantics and answer set programming have been shown to support powerful reasoning techniques such as default reasoning, counterfactual reasoning, abductive reasoning, etc. These reasoning capabilities now become available within Prolog.

The only restrictions our method places upon programs are that operands of arithmetic operations must be ground, two *negatively constrained variables* (discussed in Section 3.1.2) cannot be *disunified* with each other and left recursion cannot lead to success. Of these, the last restriction can be removed via tabling (Swift and Warren 2012). We prove the soundness of our method for all legal programs and a prototype implementation is available (Marple 2015). For convenience, we will refer to our method by the name given to its prototype implementation: **s(ASP)**.

We will begin with an overview of our method for the goal-directed execution of propositional logic programs and then examine the changes needed to adapt it to predicate logic programs containing arbitrary terms. Therefore, the remainder of the paper is structured as follows. Section 2 contains an overview of the stable model semantics as well as our propositional, query-driven execution method. Section 3 covers the expansion to predicate logic programs. Section 4 contains proofs for the correctness for our method. Section 5 provides a brief discussion of our prototype implementation along with examples of its execution. Section 7 discusses related and future work. Finally, in Section 8, we review the paper and draw conclusions.

2 Background

In this section, we will provide background information needed to understand both our method and its significance. We begin with an overview of the stable model semantics before examining our method for propositional programs, upon which our method for predicate programs is built.

2.1 The Stable Model Semantics

The stable model semantics provides an intuitive way to represent non-monotonic or common sense reasoning in normal logic programs. The building blocks of such programs are literals.

Definition 1
A **positive literal** is an atom or compound term, optionally prefixed with a hyphen ('-'). A negated or **negative literal** is a positive literal preceded by `not`. The basic term **literal** encompasses both positive and negative literals.

As we are dealing with predicate programs, it should be noted that, in the case of compound terms, literals with the same functor, arity (number of arguments) and negation are treated as instances of the same literal. For instance, `p(X)` and `p(1)` are both instances of the literal `p/1`, while `not p(1)` is an instance of `not p/1`.

Definition 2
A **normal logic program** is a program consisting of clauses of the following forms:
```
  p :- q_1, ..., qi, ..., q_m,
       not r_1, ..., not r_j, ..., not r_n.
  p :- q_1, ..., qi, ..., q_m,
       not r_1, ..., not r_j, ..., not r_n,
       not p.
  :- q_1, ..., qi, ..., q_m,
     not r_1, ..., not r_j, ..., not r_n.
```
where $m \geq 0$, $n \geq 0$, $0 \leq i \leq m$ and $0 \leq j \leq n$. Each `p`, `q`$_i$ and `not r`$_j$ is a literal. The literal to the left of the consequence operator (`:-`) is the **head** of the clause, while the literals to the right, also referred to as **goals**, form the **body**. A **rule** is the set of all clauses in a program with the same literal as their head. Either the head or body of a rule may be empty, resulting in **headless rules** and **facts**, respectively.

Unless otherwise stated, all programs referenced in this paper will be normal logic programs. Two forms of negation are permitted: *negation as failure* and *classical negation*.

Definition 3
Under **negation as failure (NAF)**, `not p` succeeds iff `p` fails.

Negative literals in the body of a rule are negated using NAF. The optional hyphen prefix indicates classical negation.

Definition 4

Under **classical negation**, -p and p cannot both be true, but it is possible for both to be false. Additionally, while not p may only appear in the body of a clause, -p may be used as the head of a clause, e.g.

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

Thus, the difference between classical negation and NAF is that classical negation can be used to define explicit rules for establishing falsehood.

Both NAF and classical negation carry the implicit constraint that a call and its negation cannot both succeed. That is, the success of p excludes not p and -p, and the success of either not p or -p will exclude p.

From an external standpoint, only positive literals may appear in the head of a rule. Internally generated dual rules, discussed in Section 3.1.4, have negated literals as their heads, but such rules can never be supplied in an input program. Note, however, that classical negation does not affect whether a literal is positive or negative. Thus -p is considered a positive literal, with not -p its negation.

The stable model semantics of a normal logic program P is defined in terms of the stable models of the program ground(P), obtained by grounding the variables in P (note that the grounding is not required to be finite). The stable models of ground(P) are traditionally identified using the Gelfond-Lifschitz method (Baral 2003). At the most basic level, the Gelfond-Lifschitz method is similar to our own: a candidate model is generated and then tested to ensure that it is a stable model.

Definition 5

A **candidate stable model** is a set of literals which is assumed to be a stable model of a given program.

Definition 6

Gelfond-Lifschitz method (GL Method): Given a grounded program P and a candidate stable model M, a residual program R is obtained by applying the following transformation rules:

1. For all literals L ∈ M, delete all clauses in P which have not L in their body.
2. Delete all remaining negated goals (of the form not X) from the bodies of the remaining clauses.

Next, the least fixed-point, F, of the residual program R is computed. If F = M, then M is a **stable model** of P.

The non-monotonicity of the stable model semantics is captured by the last two rule forms given in Definition 2. Consider the rule:

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

Following the Gelfond-Lifschitz method as outlined above, this clause restricts q (and p) to not be in the stable model (unless p happens to be in the model via another clause, in which case, due to presence of not p, this clause will be removed

```
% Given 3 birds, which can fly?
penguin(sam).        % sam is a penguin
wounded_bird(john).  % john is wounded
bird(tweety).        % tweety is just a bird

% penguins and wounded birds are still birds
bird(X) :- penguin(X).
bird(X) :- wounded_bird(X).

% penguins and wounded birds are abnormal
ab(X) :- penguin(X).
ab(X) :- wounded_bird(X).

% birds can fly if they are not abnormal
flies(X) :- bird(X), not ab(X).

% explicit closed world assumptions
-flies(X) :- ab(X).
-flies(X) :- -bird(X).

-wounded_bird(X) :- not wounded_bird(X).
-penguin(X) :- not penguin(X).
-ab(X) :- not ab(X).
-bird(X) :- not bird(X).
```

Fig. 1. A version of the classic "Tweety Bird" problem with a combination of classical negation and NAF.

while generating the residual program). Note that even though a program can have other rules to establish that `q` is in the stable model, adding the rule above forces `q` to not be in the model unless `p` succeeds through another clause, thus making the stable model semantics non-monotonic.

Sample normal logic programs are shown in Figures 1, 3 and 4. The program in Figure 1 shows an example of default reasoning with the *closed world assumption (CWA)*. Under the CWA, if we do not know a piece of information, we infer it to be false. With this program, if we pose the query `?- -flies(X)`, we should get the answers `X = sam` and `X = john`.

2.2 Our Method for Propositional Programs

Now that we have introduced the basics of the stable model semantics, we can discuss our method for goal-directed execution of propositional programs, which can be viewed as a stepping stone to our method for predicate programs. Our method for propositional programs has been proven sound and complete with respect to the Gelfond-Lifschitz method and forms the core of the Galliwasp ASP system (`http://galliwasp.sourceforge.net`) (Marple and Gupta 2013; Marple 2014b;

Marple 2014a). The two key aspects of our propositional method are its handling of rules containing odd loops over negation and its use of *coinduction*.

Both our propositional and predicate methods categorize rules by examining the call graph and checking the number of negations between any recursive calls.

Definition 7
A program's **call graph** is a directed, weighted graph with one node for each positive literal in the program. Edges are drawn from rule heads to their goals. While only positive literals are used as nodes, negation is preserved using weighted edges: edges corresponding to a positive literal are given a weight of 0, while edges corresponding to negative literals are given a weight of 1. To keep track of which rules are part of a given cycle, each edge is also paired with an ID indicating which rule produced it.

First, the call graph is traversed to identify any odd loops over negation.

Definition 8
An **odd loop over negation (OLON)** occurs when a cycle in the call graph contains an odd number of negations.

Each rule in the program is then classified using the following definitions:

Definition 9
An **OLON rule** is a rule which can be called as part of an OLON.

Definition 10
Ordinary rules have at least one path in the call graph which will not result in an odd loop over negation.

Note that rules with an empty head are always treated as OLON rules. Additionally, a rule can be both an OLON rule and an ordinary rule via different paths in the call graph.

OLON rules are important to the stable model semantics because they have the ability to place global constraints on a program. These constraints must be satisfied by any stable model, even if the OLON is never reached during execution. Consider the following two forms of OLON rules:

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

where B is some conjunction of goals. For the first rule, any stable model must satisfy one of two cases: (i) p is added to the model by another rule in the program, or (ii) at least one goal in B must fail. That is, the rule imposes the global constraint p ∨ not B. For headless rules (a shorthand for the second form), the second case must always hold, imposing the global constraint not B.

Programs are executed using a modified form of coinduction extended with negation. First, a query is extended to enforce the constraints imposed by any OLON rules in the program. Then, this query is executed using a modified form of coinductive SLD resolution (Gupta et al. 2007).

Definition 11
Under **SLD resolution** (Lloyd 1987), query is executed by calling each goal in turn. Calls are added to the call stack and expanded by selecting clauses whose head unifies with the call and recursively calling the goals in the body. A call succeeds when this expansion becomes empty (the call or its children unify with facts). If no expansion is possible, backtracking occurs: execution is rolled back to the previous expansion operation and the call is expanded using the next matching clause. A call fails when no matching clauses remain. Execution succeeds when every goal of the query has succeeded and fails when both expansion and backtracking are impossible.

Definition 12
Coinductive SLD resolution (co-SLD resolution) expands SLD-resolution by storing each succeeding call in a set called the **coinductive hypothesis set (CHS)**. If a call unifies with a call that is already in the CHS, or with an ancestor in the call stack, the call is allowed to *coinductively succeed* without further expansion (Gupta et al. 2007).

Under the stable model semantics, the condition for coinductive success via the call stack is modified such that only cycles containing *even loops* may succeed. This modification is necessary because the stable model semantics requires that *positive loops* fail, while traditional coinduction would allow them to succeed.

Definition 13
An **even loop** occurs when a recursive call is encountered with an even, non-zero number of negations between the call and its ancestor in the call stack.

Definition 14
A **positive loop** occurs when a recursive call is encountered with no negations between the call and its ancestor in the call stack.

Our methods also add the idea of *coinductive failure*, in which failure and backtracking occur if the negation of a call unifies with a call in the call stack or CHS. This ensures that the CHS remains consistent, as `p` and `not p` can never be present at the same time.

Under our methods, the CHS also serves as a *candidate partial model*, or candidate model for simplicity. These are conceptually the same as the candidate stable models described in Definition 5, except that our method focuses on finding subsets of stable models rather than complete models (see Definition 16).

Candidate partial models are generated by executing the ordinary rules in a program and then testing to ensure that they satisfy any constraints imposed by OLON rules. This testing is handled by the *non-monotonic reasoning check*.

Definition 15
The **non-monotonic reasoning check (NMR check)** is a special rule responsible for applying the constraints imposed by OLON rules. A call to the NMR check is automatically appended to each query.

For each OLON rule in a program, a "sub-check" rule with a unique head is created to apply the corresponding global constraint (discussed earlier in the section). The head of each sub-check rule is then added to the body of the NMR check. Sub-check rules are created by adding the negation of the corresponding OLON rule's head to the body (if not already present) and then negating the rule. Each clause is processed independently, so no modification is needed if a goal appears in multiple OLONs or as the head of multiple OLON rules. For instance, the rules

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

would produce the sub-check rules

```
chk_p1 :- not B.
chk_p1 :- p.
chk_p2 :- q.
chk_p2 :- p.
chk_p3 :- not q.
chk_p3 :- not r.
chk_p3 :- p.
```

As a result, if the NMR check succeeds, the candidate partial model in the CHS must satisfy every OLON rule in the program. Correspondingly, if a program or candidate partial model is inconsistent, the NMR check will trigger failure and backtracking. For example, a program containing the rule

```
:- not c.
```

where `c` does not appear anywhere else in the program will have no stable model. The method enforces this by creating the NMR sub-check

```
chk :- c.
```

Since the program contains no rules for `c`, the check is unsatisfiable and execution will eventually fail.

Upon successful execution of both the query and NMR check, the CHS will be returned as a partial stable model.

Definition 16
A **partial stable model** is a set of literals which is guaranteed to be a subset of some stable model of the program (Marple et al. 2012).

3 The s(ASP) Method

Now that we have introduced the stable model semantics and our goal-directed method for propositional programs, we can discuss our predicate method. We will begin with some fundamentals before looking at several key aspects of the method itself. Finally, we will give an overview of the completed method.

3.1 s(ASP) Fundamentals

Before we can discuss the actual execution of our method, we must first introduce a number of core concepts. These include s(ASP)'s universe, its system of variables and constraints, its use of constructive negation and its restrictions on legal programs.

3.1.1 The s(ASP) Universe

One of the defining aspects of s(ASP) is its universe. While most logic programming semantics utilize the Herbrand universe, it is insufficient for our purposes. While the Herbrand universe may be finite, s(ASP) explicitly requires that its universe always be infinite, the reasons for which will be discussed in Section 3.1.2. To ensure that this property always holds, we rely on a universe defined as follows:

Definition 17
The **s(ASP) universe**, U_S, is an infinite, proper *superset* of the Herbrand universe.

Formally, this is achieved by extending the language of propositional stable models with an infinite number of special constants in the manner of Shoenfield (Shoenfield 1967, p. 46) Shoenfield works with first-order mathematical logic rather than non-monotonic logic programming, but translation between the two is straightforward: Shoenfield's definition of a first order theory corresponds to a system consisting of a language, its universe, a semantics, and a program to be executed. As Shoenfield proves, a first order theory whose language is extended using his special constants technique is a "conservative extension" of the original theory. That is, a formula (rule) from the original theory will hold in the extended theory iff it holds in the original theory. In simpler terms, *our extension of the universe does not affect the correctness of programs which have been grounded over the Herbrand Universe.* Thus, for simplicity, subsequent references to the stable model semantics of propositional programs and the GL method will refer to variants which have been extended to use the s(ASP) universe.

3.1.2 Variables and Constraints

The most obvious step in supporting predicate programs is to accommodate variables. Additionally, the constructive negation employed by s(ASP) relies on extending variables with simple constraints. In turn, unification and *disunification* must be modified to accept such variables.

Instead of the traditional states of bound and unbound, s(ASP) variables can be either bound or negatively constrained.

Definition 18
A **negatively constrained variable** is associated with a *prohibited value list*—a list of prohibited values—and represents the set of all values in the s(ASP) universe which are not in this list.

Thus, if the prohibited value list of a variable X contains the constants a and b, then X may be bound to any value *except* a and b. Unbound variables are treated as a special case of negatively constrained variables in which the prohibited value list is empty. Note that values in a prohibited value list need not be ground, but must be at least partially bound. The presence of a negatively constrained variable in a prohibited value list would violate our restriction against disunifying two negatively constrained variables, discussed further in Section 3.1.5.

As discussed in Section 3.1.1, the s(ASP) universe is an infinite superset of the Herbrand universe. The discrepancy between variables with finite domains and variables defined in terms of the s(ASP) universe can lead to what we will refer to as empty variables.

Definition 19
A variable which is empty with respect to some domain D, referred to as an **empty variable** for simplicity, is a negatively constrained variable whose prohibited value list contains every element of D.

This brings us to the reason that s(ASP) requires a special universe. Clearly, were a program to be grounded over some D, no value could be assigned to such a variable. However, *it is impossible to ground a program over the s(ASP) universe.* Because it is an infinite, proper superset of the Herbrand universe, the s(ASP) universe will always contain elements which are not present in a given grounding. This provides us with several important properties:

Proposition 1 (*Properties of s(ASP) Variables*)
1. It is impossible for a legal program to negatively constrain a variable against every element of the s(ASP) universe.
2. A variable can never be empty with respect to the s(ASP) universe itself.
3. The domain of a negatively constrained variable defined in terms of the s(ASP) universe will always be *infinite*.

Proof
All of these properties may be derived directly from Definition 17, Definition 19 and our restriction against disunifying two negatively constrained variables (Section 3.1.5).   □

As a result of these properties, variables which are empty with respect to the Herbrand universe, or some subset of it, do not trigger failure. For example, consider the following program, with rules for negation (termed dual rules, discussed in Section 3.1.4) added for clarity:

```
d(1).
p(X) :- not d(X).
not d(X) :- X \= 1.
not p(X) :- d(X).
```

If we assume D to be {1}, then a grounding of this program would be

```
d(1).
p(1) :- not d(1).
not d(1) :- 1 \= 1.
not p(1) :- d(1).
```

Given the query `?- p(X).` the grounded program will always fail. However, execution of the predicate program with s(ASP) will succeed, returning the solution { `p(X)`, `not d(X)` (`X \= 1`) }.

While results involving empty variables may be different from those of a corresponding grounded program, the two will never be inconsistent with each other. Instead, they simply convey different, equally correct information. The failure of the grounded program indicates that no solution exists for the domain {1}, while the success of the predicate program indicates that a solution would exist if the domain were to be extended.

3.1.3 Constructive Unification and Disunification

Now that we have introduced negatively constrained variables, unification and disunification must be extended to work with them. To differentiate the modified versions from the originals, we will refer to them as *constructive* unification and disunification. For cases where neither argument contains a negatively constrained variable, the constructive algorithms are identical to the traditional ones.

Definition 20
The cases for **constructive unification** are as follows:

- Constructive unification of a negatively constrained variable with a non-variable value will succeed if the non-variable value does not constructively unify with any element in the variable's prohibited value list.
- Constructive unification of two negatively constrained variables will always succeed, setting their shared prohibited value list to the union of their original lists.
- Constructive unification of two compound terms is performed recursively: first, the functors and arities are tested, then each pair of corresponding arguments is constructively unified.
- In cases where neither argument contains a negatively constrained variable, the result is identical to that of traditional unification.

Thus, if the prohibited value list of `X` contains `a`, and the prohibited value list of `Y` contains `b`, then unifying them, `X = Y` where `=` is the unification operator, will extend the prohibited value list of both variables to `[a, b]`. After the operation, any subsequent attempt to unify either variable with `a` or `b` will fail.

Definition 21
Constructive disunification is the dual of constructive unification with one exception: in accordance with the restrictions given in Section 3.1.5, constructive disunification of two negatively constrained variables will produce an error. The remaining cases are as follows:

- Constructive disunification of a negatively constrained variable and a non-variable value will always succeed, adding the non-variable value to the variable's prohibited value list.
- Constructive disunification of two compound terms is performed by first testing functors and arities. If either of these does not match, the operation succeeds deterministically. Otherwise, the pairs of corresponding arguments are disunified recursively. *Non-deterministic* success occurs as soon as the operation succeeds for a pair of arguments, with subsequent pairs tested upon backtracking.
- In cases where neither argument contains a negatively constrained variable, the result is identical to that of traditional disunification.

Given a variable `X` whose prohibited value list contains `a`, disunifying `X` with a constant `c`, i.e., solving `X \= c` where `\=` represents the disunification operator, will extend the prohibited value list of `X` to `[a,c]`, i.e., `X` cannot be bound to either `a` or `c`. Under our program restrictions, discussed further in Section 3.1.5, the disunification of two negatively constrained variables is considered illegal. There is, however, an exception to this rule involving variables in even loops, discussed in Section 3.2.2.

Note that constructive disunification of compound terms has the potential to be non-deterministic. Consider the following statement:

```
a(X, Y) \= a(1, 2)
```

This operation can succeed for both `X \= 1` and `Y \= 2`, but applying both constraints at the same time would result in incompleteness, excluding cases such as `a(1, 3)`. To preserve correctness, the operation will succeed non-deterministically, first succeeding for `X \= 1` and then for `Y \= 2` upon backtracking.

By construction, constructive unification and disunification are sound and complete with respect to their traditional counterparts for all legal cases. However, as constructive disunification of compound terms may be non-deterministic, backtracking may be required to produce all values.

3.1.4 Constructive Negation and Dual Rules

One of the cornerstones of s(ASP) is constructive negation. The stable model semantics relies on negation as failure (NAF), which normally returns no bindings or other information from a failed call. However, a goal-directed implementation of the stable model semantics needs to know *why* a call failed rather than simply that it failed. With constructive negation, negated calls can be treated the same way as non-negated calls, binding variables and contributing to the model. To implement constructive negation, s(ASP) computes the *completion* of the program, extending it with rules for negative information (Lloyd 1987). These new rules are called *dual rules*, as they represent the negations, or duals, of the rules in the original program (Alferes et al. 2004).

Definition 22
Dual rules are rules for the negation of a predicate which will succeed whenever a call to the original predicate would fail under NAF.

In cases without variables or side effects, a predicate's dual rule can be computed by simply applying DeMorgan's laws:

$$\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q) \tag{1}$$

$$\neg(P \vee Q) \Leftrightarrow (\neg P) \wedge (\neg Q) \tag{2}$$

For example, given a predicate `p` with the clauses

```
p :- a, not b.
p :- r.
```

the dual of `p` would be

```
not p :- np1, np2.
np1 :- not a.
np1 :- b.
np2 :- not r.
```

Although the above method works for propositional programs, it requires modification to work with predicate programs. To begin with, we will examine two cases that must be addressed to account for the introduction of unification. First, unification is performed between each of a call's arguments and the corresponding arguments in the head of the selected clause. For example, consider a call `t(X, 2)` and a clause with the head `t(A, B)`. When expanding the call using this clause, `X` will be unified with `A` and `B` will be unified with 2. Second, the presence of variables prevents the goals in a dual rule from being considered independently. Consider the clause

```
p(X) :- q(X), r(X).
```

Our above method would produce the dual

```
not p(X) :- not q(X).
not p(X) :- not r(X).
```

However, `q(X)` can modify the value of `X`, affecting the outcome of `r(X)` and, by extension, `not r(X)`. Correct dual rules must account for this. Finally, we will look at how to handle the implicit quantifiers on each variable in a clause.

While the unification that occurs between a call and a rule's head when expanding a call is easy to overlook, dual rules must account for it. Since a dual rule must succeed when the original would fail, it must also succeed when these unification operations would fail. We address this by abstracting such operations to remove them from the head and make them explicit. Before computing the dual of a clause, the arguments in the head are examined. If a variable occurs more than once in the head of a clause, each occurrence after the first is replaced with a new variable and a goal unifying the original variable and the new one is added to the beginning of the

rule body. If an argument is a non-variable, it is replaced with a new variable and a goal unifying the new variable to the non-variable value is added to the beginning of the rule body. So, the rule

```
t(A, A).
```

would first be transformed into

```
t(A, B) :- A = B.
```

which would produce the dual rule

```
not t(A, B) :- A \= B.
```

The operations are the same for compound terms, except that they are performed recursively on the arguments of the terms.

Unification also prevents us from considering each goal in a clause independently when constructing duals. Because each goal has the potential to alter any non-ground variable that it is called with, each goal in a clause may depend on the goals called before it. Therefore, before calling the negated goal itself, each dual clause must call any goals on which the current goal depends. While it should be possible to determine the dependencies of a given goal, our current strategy is to simply include in each clause every goal prior to the current one. This means that the rule

```
p(X, Y) :- not q(X), t(Y, Y).
```

would produce the dual rule

```
not p(X, Y) :- q(X).
not p(X, Y) :- not q(X), not t(Y, Y).
```

Our algorithm for computing dual rules requires one more modification to properly handle predicate programs. This is due to the fact that variables have implicit quantifiers: variables in the head of a clause are universally quantified, while variables in the body (body variables) are existentially quantified. Thus, the clause

```
q(X) :- not p(X, Y).
```

is equivalent to the formula

$$\forall X (q(X) \leftarrow \exists Y \neg p(X, Y))$$

Duals of clauses that contain variables must negate these quantifiers as well. The universal and existential quantifiers are duals of each other, so the dual of the above formula is

$$\forall X (\neg(q(X)) \leftarrow \neg(\exists Y \neg p(X, Y)))$$
$$\equiv \forall X (\neg q(X) \leftarrow \forall Y \neg(\neg p(X, Y)))$$
$$\equiv \forall X (\neg q(X) \leftarrow \forall Y p(X, Y))$$

Definition 23
Body variables are variables which occur in the body of a clause, but not in its head.

As this example shows, any body variables in a clause will be universally quantified in the clause's dual. That is, for some binding of the head variables, the clause must succeed for all values (or combinations of values) of the body variables. To create such duals, we developed a special for-all mechanism which relies on negatively constrained variables.

Definition 24
A **for-all** is a s(ASP) goal of the form `forall(V, G)` where `V` is a variable and `G` is a goal. A for-all succeeds if `G` succeeds for all values of `V`.

These for-alls are used to create dual rules that universally quantify body variables from the original rule. First the dual rules for a clause are computed as if no body variables were present, except that the predicate in the head is replaced with a new, unique one. Next, the body variables are added to the head of each dual. Finally, a clause for the dual is created containing a for-all over the new predicate. So, the rule

```
q(X) :- not p(X, Y).
```

would produce the dual rule

```
not q(X) :- forall(Y, nq1(X, Y)).
nq1(X, Y) :- p(X, Y).
```

Note that multiple body variables can be handled by nesting for-alls. For example `forall(X, forall(Y, p(X, Y)))` will succeed if `p(X, Y)` succeeds for all values of `X` and `Y`.

At runtime, a for-all is executed by calling `G` with `V` unbound. If `G` succeeds, the value of `V` is checked. If `V` is still unbound, `G` is added to the CHS and the for-all succeeds. If `V` is bound, failure and backtracking take place. However, if `V` is negatively constrained, `G` is added to the CHS and then called again for each value in `V`'s prohibited value list, substituting the constrained value for `V`. If `G` succeeds for every constrained value, the for-all succeeds and `G` is added to the CHS with `V` unbound. Consider the following program with dual rules added:

```
p :- not q(X).
q(Y) :- Y = a.
q(Y) :- Y \= a.

not p :- forall(X, np1(X)).
np1(X) :- q(X).

not q(Y) :- nq1(Y), nq2(Y).
nq1(Y) :- Y \= a.
nq2(Y) :- Y = a.
```

and the query `?- not p`. The clause for `not p` will execute `forall(X, np1(X))`, which will in turn execute `q(X)`. The first clause of `q(Y)` will succeed, unifying `Y` with `a`. However, because `X` will be bound when `np1(X)` succeeds, execution will fail and backtrack, forcing the second clause for `q(Y)` to be selected. This clause will succeed, adding `a` to `Y`'s prohibited value list. Because `X` is now constrained when `np1(X)` succeeds, the forall will call `np1(X)` substituting each member of `X`'s prohibited value list for `X` in turn. In this case, the only extra call will be `np1(a)`. The call to `q(a)` will succeed via the first clause for `q(Y)`, causing the call `np1(a)` to succeed. Finally, since `np1(X)` has succeeded both with `X` constrained and for every member of `X`'s prohibited value list, the forall itself will succeed, setting `X`'s prohibited value list to an empty list before adding it to the CHS.

Two points are worth noting. First, for-alls are only used in internally generated code; they are not made available to users. This allows us to ensure that `V` will always be unbound at the time the call is made. Second, the correctness of our for-all algorithm relies on our use of the s(ASP) universe, and specifically on the properties given in Proposition 1. The above algorithm will not work in cases where the domain of a variable may be finite or where it is possible to constrain a variable against every element of its domain. For example, given the rule

```
p(1).
```

the goal `forall(X, p(X))` will fail under our method. This behavior is correct when the domain of `X` is the s(ASP) universe, as required by our method. However, were the domain of `X` allowed to be $\{1\}$, the failure of the forall would be incorrect.

It is important to note that our use of dual rules may sometimes produce unexpected results. Because the disunification of two negatively constrained variables is illegal under our program restrictions (Section 3.1.5), it is implicitly illegal to call the dual of a rule which contains a non-ground structure in the head, unless the corresponding argument in the call is structured such that no comparisons between a variable and a non-ground element are made. Since the argument in the call and the structure in the head of the rule are unified in the original clause, the dual will contain a disunification between these elements. Consider the rule:

```
p([X | T]) :- q(X), p(T).
```

Because X and T are present in a structure in the head, the first step in creating a dual would be to abstract them out of the head, as discussed earlier in this section. However, that makes them body variables, necessitating the use of a forall. Thus, the final dual is:

```
not p(Z) :- forall(X, forall(T, np1(Z, X, T))).
np1(Z, X, T) :- Z \= [X | T].
np1(Z, X, T) :- Z = [X | T], not q(X).
np1(Z, X, T) :- Z = [X | T], q(X), not p(T).
```

If this dual is called with a list containing unbound variables, the disunification operation in the first clause of `np1(Z, X, T)` may encounter illegal cases and trigger a fatal error. For instance, given the call `not p([A | B])`, the corresponding call

will be `np1([A | B], X, T)`. The disunification operation in the first clause of
np1(Z, X, T) will attempt to recursively disunify `A` with `X` and `B` with `T`. As both
operations are illegal, execution will halt and report an error. The same situation
will occur if the same variable occurs more than once in the head of a rule. Because
subsequent occurrences are replaced with unique variables and goals are added to
the rule body unifying them with the original, the dual contains corresponding
disunification operations. As a result, calling the dual with non-ground values for
such variables will trigger an error. Consider the following rule and its dual:

```
t(A, A).
not t(A, B) :- A \= B.
```

Because the original rule contains no disunification operations, `t(A, A)` can be
called for any value of `A`. However, the dual will produce an error unless either `A`
or `B` is bound such that no negatively constrained variables are disunified. Consider
the following query:

```
?- not t(A,2), not t(B,1), not t(A,B).
```

The call `not t(A, 2)` will succeed constraining `A` against 2 and the call `not t(B,
1)` will succeed constraining `B` against 1. However, the call `not t(A, B)` will trigger
an error when it attempts to disunify `A` and `B`.

3.1.5 Restrictions on Legal Programs

While one of our primary goals in developing s(ASP) has been to support the largest
possible class of normal logic programs, some restrictions are still required under
our current method. Now that we have introduced the other core concepts, we are
equipped to discuss these restrictions.

Definition 25
A **legal program** is a normal logic program, as defined in Definition 2, which
satisfies the following restrictions:

1. Operands of arithmetic operations must be ground at the time they are executed.
2. Left recursion cannot lead to success.
3. A negatively constrained variable (see Section 3.1.2), cannot be *disunified* with or *constrained against* another negatively constrained variable.

Of these restrictions, we believe that the first two may eventually be lifted by
modifying our method. Delayed expansion and improved constraint handling may
remove the need for ground arithmetic, while tabling should eliminate issues related
to left recursion. At present, these changes are left for future work.

The remaining restriction is integral to our current method. Recall from Proposition 1 that one of the key properties of our method is that the domain of a negatively
constrained variable can never be empty with respect to the s(ASP) universe. Allowing a negatively constrained variable to be constrained against another with
an empty prohibited value list would violate the property, breaking our method.

Improved constraint handling might allow variable disunification to occur in cases where neither variable's domain would become empty, but we leave this for future work. For the time being, the only exception to this restriction is a special case involving even loops, described in Section 3.2.2.

3.2 Constructing the s(ASP) Method

Now that we have covered the necessary core concepts, we can discuss several other important aspects of the s(ASP) method. To avoid repetition, we will discuss these aspects in terms of differences between our propositional method, described in Section 2.2, and our predicate method. Thus, in this section, we will look at changes made to coinduction (Section 3.2.1), even loops (Section 3.2.2) and consistency checking (Section 3.2.3).

3.2.1 Coinduction

Like our propositional method, s(ASP) executes programs using a modified form of co-SLD resolution (Gupta et al. 2007). The most obvious changes to the predicate method are the incorporation of negatively constrained variables and the corresponding use of constructive unification and disunification. However, additional changes are needed to handle negatively constrained variables correctly. In this section, we will look at the changes needed to adapt both coinductive failure and coinductive success for use with s(ASP).

Under our method for propositional programs, coinductive failure occurs when the negation of a goal is present in the CHS (Marple et al. 2012). For example, if `not p` is found in the CHS when checking `p`, the call to `p` will fail coinductively. So, when dealing with propositional programs, it is sufficient to simply fail when a match for the negation is found. It might seem that extending this method to the predicate case would be as simple as checking to see if a goal's negation unifies with any entry in the CHS. However, this can lead to incorrect behavior when combined with CHS entries that include negatively constrained variables. Consider a program consisting of the following rule, with its dual added for convenience:

```
pi(X) :- X = 3.14.
not pi(X) :- X \= 3.14.
```

Correct behavior requires that a call to `not pi(X)` should always succeed with X \= 3.14. However, this may not happen if we rely on ordinary unification to check for coinductive failure. For instance, if we execute the program with the query `?- pi(Y), not pi(X)`, the goal `pi(Y)` will succeed for Y = 3.14 and execution will move on to `not pi(X)`. However, the negation of `not pi(X)` will unify with the CHS entry for `pi(Y)`, causing the query to incorrectly fail.

Our solution to the problem once again relies on constructive negation. Instead of simply failing when the negation of a goal unifies with an entry in the CHS, coinductive failure is viewed as a filter, allowing bindings whose negation does not unify with a CHS entry to succeed. This is accomplished by constraining variables

in the call such that the call's negation will no longer unify with any entry in the CHS. In the above example, when `not pi(X)` is tested, X will be constrained against 3.14, preventing its negation, `pi(X)`, from unifying with the CHS entry for `pi(3.14)`.

Two things are worth noting about this process. First, the *negation* of a goal is checked rather than its *dual*. For example, the dual of `pi(3.14)` is `not pi(X)` where X is constrained against 3.14, but the negation of `pi(3.14)` is simply `not pi(3.14)`. Second, the coinductive failure check may be non-deterministic. Consider the following rule:

```
q(X) :- X \= 2, X \= 3.
```

Given the query `?- q(X), not q(Y)`, at the time `not q(Y)` is called, the CHS will contain `q(X)`, with X constrained against both 2 and 3. This leaves two ways for the coinductive failure check to succeed: Y may be set to either 2 or 3. This choice will be made non-deterministically: 2 will be selected first, but 3 may be chosen when backtracking.

Non-determinism may also arise when testing goals with more than one argument. Consider a modification of the previous rule:

```
q(X, Y) :- X \= 2, Y \= 3.
```

Now, given the query `?- q(X, Y), not q(A, B)`, at the time `not q(A, B)` is called, the CHS will contain `q(X, Y)`, with X \= 2 and Y \= 3. Again, this leaves two ways for the coinductive failure check to succeed: A may be set to 2 or B may be set to 3.

To ensure that all cases are considered when executing the coinductive failure check, each argument is considered separately, first to last, with subsequent arguments being selected on backtracking once all choices for the previous argument have been exhausted. Consider the rules:

```
q(W, X) :- W \= 2, W \= 3, X \= 2, X \= 3.
q(Y, 3) :- Y \= 2, Y \= 3.
```

Now, given the query `?- q(W, X), q(Y, 3), not q(A, B)`, at the time `not q(A, B)` is called, the CHS will contain both `q(W, X)`, with W \= 2, W \= 3, X \= 2, X \= 3 and `q(Y, 3)`, with Y \= 2, Y \= 3. When executing the coinductive failure check for `not q(A, B)`, the first argument will be examined first, and the goal will be allowed to succeed first with A = 2 and then with A = 3. Having exhausted all options for success via the first argument, further backtracking will lead to the second argument being examined. Here, success is possible only when B = 2. Thus, the coinductive failure check for `not q(A, B)` will succeed up to three times: once each for A = 2, A = 3 and B = 2. Together, these cases cover all possible scenarios where `not q(A, B)` does not unify with any element of the CHS.

Like coinductive failure, coinductive success must be also modified to work with predicate programs. In this case, we must differentiate between CHS entries and ancestors in the call stack which are exact matches for a call and those which simply unify with it.

Definition 26

Two terms are an **exact match** if they can be constructively unified without altering the prohibited value lists of any variables present in either argument.

For example, given variables X and Y, if both are constrained against 2, then they are an exact match. However, if X is constrained against 2 and Y against 2 and 3, they are unifiable, but not an exact match. The same applies to compound terms: `f(X)` and `f(Y)` are an exact match if X and Y are an exact match.

When testing for coinductive success, exact matches will allow success or failure to be deterministic, while other matches will be non-deterministic. The testing process for a call C is as follows:

1. C is tested against the CHS for exact matches. If one is found, deterministically succeed.
2. C is tested against each entry D in the call stack. The number of negations between C and D are counted, excluding C and D themselves.

   (a) If C and D are an exact match with no intervening negations, fail deterministically.
   (b) If C constructively unifies with D with an even, non-zero number of intervening negations, succeed. If C and D are an exact match, success is deterministic, otherwise it is non-deterministic. This non-determinism simply allows C to be executed normally by step 3 upon backtracking, allowing solutions which might otherwise be missed. *Unification here requires an occurs check for correctness (see below).*

3. If no matches are found or all deterministic matches have been exhausted, execute C normally.

The use of exact matches in steps 1 and 2(a) is necessary for completeness. With constructive unification, a call would always succeed if it unified an entry in the CHS and fail if it unified with an entry in the call stack with no intervening negations. However, this behavior could result in solutions being skipped. For example, in step 2(a), exact matches are needed to avoid false positives when detecting positive loops. Consider the rules

```
r(V) :- r(V2).
r(3.14).
```

Were a program containing these rules to be grounded, positive loops would only be present for those cases where V = V2. However, since V2 will always be unbound, `r(V)` and `r(V2)` will always constructively unify. As a result, a positive loop would always be detected if ordinary coinductive success were used, leading to failure in all cases. However, if exact matches are required, a positive loop will be detected only when V is unbound. As a result, the query `?- r(V)` will succeed for any V: a call with an unbound V will succeed via the second rule, while call with a ground or semi-ground V will succeed via the first, with `r(V2)` being satisfied by the second. Consider the query `?- r(1)`:

1. `r(1)` will be checked for coinductive failure/success.
2. Since the conditions for immediate success or failure are unmet, `r(1)` will be added to the call stack and expanded using the first rule.
3. `r(V2)` will be checked for coinductive failure/success.
4. Since the conditions immediate success or failure are unmet, `r(V2)` will be added to the call stack and expanded using the first rule.
5. The new `r(V2)`, `r(V2')` for convenience, will be checked for coinductive failure/success.
6. Since `V2'` and `V2` both have empty prohibited value lists, `r(V2')` and `r(V2)` are an exact match. As there are no intervening negations between the two calls, `r(V2')` will fail.
7. Execution will backtrack to the expansion of `r(V2)` and try the second rule.
8. `r(V2)` will unify with `r(3.14)` and succeed.
9. `r(1)` will succeed, returning the partial stable model $\{$`r(1)`, `r(3.14)`$\}$.

The criteria for step 2(b) is similar to that of our propositional method: coinductive success only occurs if an even, non-zero number of negations exist between a recursive call and its ancestor in the call stack (Marple et al. 2012). However, when attempting to unify the current call with an ancestor in the call stack, an *occurs check* is necessary to retain correctness. That is, a variable cannot unify with any term which contains the same variable. Consider the following rules for successor notation:

```
n(0).
n(s(X)) :- n(X).
```

Given the query ?- `n(s(s(X)))`. s(ASP) will produce the partial stable models $\{$`n(0),n(s(0)),n(s(s(0)))`$\}$, $\{$`n(0),n(s(0)),n(s(s(0))),n(s(s(s(0))))`$\}$ and so on. However, without an occurs check, the query would fail: `n(s(s(X)))` would be added to the call stack and `n(s(X))` would be called. However, when checking coinductive success, `n(s(X))` would unify with `n(s(s(X)))`, triggering failure due to the lack of intervening negations. The occurs check prevents the two calls from unifying, thereby preserving correctness.

It is also important to note that coinduction will prevent infinite looping unless unbounded recursion occurs in such a way that no recursive call can constructively unify with any of its ancestors. For instance, the following program will produce infinite looping:

```
s(X) :- X2 is X + 1, s(X2).
?- s(1).
```

In this example, every instance of `s(X)` will be ground and unique, thus preventing coinductive success from ever occurring. However, such cases can be avoided by rewriting the rules in question to ensure that either a recursive call will eventually unify with some ancestor or that some base case will eventually be reached.

3.2.2 Even Loops

Even loops (Definition 13) have special significance in the stable model semantics: they indicate that a goal may be either true or false. Under s(ASP), a special case of even loops must be addressed for the sake of completeness: even loops containing loop variables.

Definition 27
Loop variables are variables which occur in both a recursive call and its ancestor in an even loop, and are unbound or negatively constrained when the ancestor call succeeds.

Observe that for a variable to be present in both a recursive call and its ancestor, it must also occur in each of the intervening calls which are part of the even loop. In this situation, the chain of literals in the loop may be either true or false for every grounding of the loop variables which does not produce a contradiction in the CHS. Because the s(ASP) universe is infinite, the result is that any program with at least one stable model containing a loop variable will have an infinite number of stable models which contain an infinite number of elements, and thus have an infinite number of partial stable models. Consider the following program:

```
p(X, Y) :- not q(X, Y), t(Y, Y).
q(X, Y) :- not p(X, Y).
```

with the query ?- q(X, Y). Because X and Y occur as loop variables, q(X, Y) and not p(X, Y) may be true or false for each possible grounding of X and Y, so long as their truth values are the same for a given combination (opposing truth values would produce a contradiction). For example, {q(1, 2), not p(1, 2)} and {p(a, b), q(1, 2), not p(1, 2), not q(a, b)} are both partial stable models of the above program. Furthermore, we know from the variable properties given in Proposition 1 that X and Y's domains are infinite, so there must be an infinite number of these partial stable models.

With this in mind, completeness requires that we have a means of representing the potentially infinite number of partial stable models which may result from loop variables. Our mechanism for this is to prefix loop variables with a question mark (?) when printing them, indicating that literals containing them may be either true or false for each grounding of the loop variable which does not produce a contradiction. Observe that no information will be lost: it will always be possible to ground our output such that it will produce a subset of any given stable model for which the query would succeed. For example, for the query ?- q(X, Y), the above program will produce a single result, {q(?X, ?Y), not p(?X, ?Y)}, compactly representing an infinite number of partial stable models for each full stable model for which the query succeeds. Note that a loop variable will become an ordinary negatively constrained variable if used in a forall: having succeeded for all values, it can no longer be assigned false for any of them.

Finally, as mentioned in Section 3.1.3, loop variables force us to make an exception to the prohibition that two negatively constrained variables cannot be constrained

against each other. Ordinarily, goals in the CHS cannot be further modified, and this includes placing further constraints upon any of their variables. However, in the case of loop variables, it is necessary to ensure that no grounding allowed by the output will be incorrect. Therefore, if a call `C` is added to the CHS with a loop variable and a call for `not C` later succeeds, correctness requires that the loop variable in `C`'s CHS entry be constrained against the corresponding entry in `not C`'s entry. If the domain of a loop variable is empty when the query and NMR check succeed, failure and backtracking must occur to ensure correctness.

3.2.3 Consistency Checking

Recall from Section 2.2 that any constraints imposed by OLON rules in a program are enforced by appending a special rule, the NMR check, to each query. The NMR check calls sub-checks for each OLON rule in the program, enforcing the constraints which they impose. Two changes are necessary to support predicate programs. First, the NMR sub-checks must be generated using our new dual rule algorithm. Second, because OLON rules apply global constraints, we must ensure that the sub-checks are satisfied for all possible values of their variables.

The first step is to use our new dual rule algorithm, detailed in Section 3.1.4, when generating NMR sub-checks. As explained in Section 2.2, a sub-check is created by first appending the negation of an OLON rule's head to its body (if not already present), taking the dual of the modified rule and assigning it a unique head. Thus, with our new dual rule algorithm, the rule

```
p(X) :- q(X), not p(X).
```

will produce the sub-check

```
chk_p(X) :- not q(X).
chk_p(X) :- q(X), p(X).
```

The second issue is that the constraints imposed by OLON rules are global, so the resulting sub-checks must hold for all possible bindings of their variables. This is accomplished using our for-all mechanism, also described in Section 3.1.4. The body of the NMR check is modified by considering each variable in a sub-check goal to be universally quantified, and abstracting it with a forall. Thus, the NMR check for the above sub-check would be

```
nmr_check :- forall(X, chk_p(X)).
```

In the case of headless rules, any variables are already body variables, so the necessary forall wrappers will be applied when the sub-checks are created.

With these modifications, the NMR check will correctly apply any constraints imposed upon the program. However, this does make it more difficult to identify whether a program is legal without running it. Because each sub-check is a dual rule, the same caveats which apply to calling negated goals, discussed in Section 3.1.4, also apply to OLON rules in general.

It is interesting to note that one aspect of our method can remain almost unchanged: the initial detection of OLON rules. As explained in Section 2.2, OLON rules are detected by finding cycles in the call graph which contain an odd number of negations. At a glance, the addition of variables would appear to complicate this procedure. Consider the following rule:

```
p(X) :- q(X, Y), not p(Y).
```

An odd loop is present, but only when X = Y. It is easy to assume that such a case might require a constraint to be added so that the resulting sub-check will exclude cases where X \= Y, but this is actually unnecessary. In fact, variables and constraints can be ignored entirely when detecting OLON rules. Thus, the above rule will produce the following NMR check and sub-checks:

```
nmr_check :- forall(X, chk_p(X)).


chk_p(X) :- forall(Y, chk_p2(X,Y)).


chk_p2(X,Y) :- not q(X,Y).
chk_p2(X,Y) :- q(X,Y), p(Y).
chk_p2(X,Y) :- q(X,Y), not p(Y), p(X).
```

Observe that when X \= Y, the sub-check will always succeed: one of the first two clauses for chk_p2(X, Y) will succeed in cases where the original rule for p(X) would fail and the third clause will succeed in cases where the original rule would succeed. Indeed, this will always be the case for such "conditional" OLON rules: in cases where no OLON is present, the corresponding sub-check will always be satisfied. Therefore, while adding constraints to exclude non-OLON cases might improve performance, they are not required for correctness.

3.3 Overview of the Completed Method

Now that we have looked at the individual components, we can examine the method as a whole.

Under the **s(ASP) method**, a legal program P is executed with a query Q as follows: First, the call graph of P is examined to create the NMR check and sub-checks. Next, the body of the NMR check is appended to Q. Then, each goal G in Q is executed in order. If G is an arithmetic expression, a unification or disunification operation, or a forall, it is executed accordingly. Otherwise, G is checked against the CHS:

- If the CHS contains an exact match for not G, G fails *deterministically*.
- If the CHS contains an exact match for G, G succeeds *deterministically*.
- If no exact match is present in the CHS, G is constrained against any CHS entries which unify with not G. This process may be *non-deterministic*.

Should G pass the CHS check without succeeding or failing, the call stack is examined for any cycles containing G, starting at the most recent call and working back:

- If G is an exact match for an entry in the call stack with no intervening negations, G fails *deterministically* (positive loop).
- If G is an exact match for an entry in the call stack with an even, non-zero number of negations, G succeeds *deterministically* (coinductive success).
- If G *constructively unifies* with an entry in the call stack with an even, non-zero number of negations, G succeeds *non-deterministically* (coinductive success).

Should G pass the call stack check without succeeding or failing, either G does not match any entry in the call stack or all non-deterministic options have been exhausted. Then, G is expanded using the rules in P. If G succeeds in this manner, it is added to the CHS. If every goal in Q succeeds, the query succeeds. Finally, the domains of any loop variables are checked to ensure that they are non-empty. If so, execution succeeds and the elements in the CHS form a partial stable model of P. Figure 2 contains an abstract meta-interpreter for this method.

4 Correctness

In this section, we will discuss the correctness of the s(ASP) method. We will show that s(ASP) is sound for all legal programs (Definition 25) and argue that, while completeness is in fact impossible, the method is still useful for the vast majority of practical programs. We will begin by looking at the propositional case and then move on to the predicate case.

4.1 Propositional Programs

In the proofs below, we demonstrate that s(ASP) is sound for all legal programs grounded over the s(ASP) universe and complete for the all finite, ground, legal programs. Note that this completeness class includes finitely groundable programs which have been grounded over the Herbrand universe prior to execution.

Theorem 1
For legal, finite, ground programs, our predicate method is equivalent to our propositional method.

Proof
This theorem holds by design. Our propositional method forms the basis for our predicate method, and all the modifications we have introduced apply only to cases involving variables and illegal programs. Thus, for a legal, finite, ground program, which, by definition, contains no variables, the two methods will be equivalent. □

Corollary 1
Our predicate method is sound and complete for finite, legal, ground programs.

Proof
Our ground method has already been proven sound and complete for the set of all finite programs grounded over the Herbrand universe (Marple et al. 2012). Therefore, by Theorem 1, our predicate method is also sound and complete for these programs. □

```
sasp(Q, NMR) :-
    append(Q, NMR, Q2),
    exec_goals(Q2),
    check_loop_variable_domains,
    print_chs.

exec_goals([X | T]) :-
    exec_goal(X),
    exec_goals(T).
exec_goals([]).

exec_goal(X) :-
    X = forall(V, G), !,
    exec_forall(V, G).
exec_goal(X) :-
    check_chs_and_call_stack(X, CHSResult),
    exec_goal2(X, CHSResult).

exec_goal2(X, success). % coinductive success
exec_goal2(X, expand) :- % expand using rules
    get_matching_rule(X, R), % select subsequent rules on backtracking
    exec_goals(body(R)),
    add_to_chs(X).

exec_forall(V, G) :-
    unbound(V), % fail if variable is bound or constrained
    exec_goal(G), % first solve goal normally
    Cons = get_constraints(V), % fail if variable is bound
    exec_with_each_constraint_value(G, V, Cons),
    set_unbound(V), % goal succeeded for all V.
    add_to_chs(G).

check_chs_and_call_stack(X, failure) :-
    Xn = dual(X),
    exact_match_in_chs(Xn), !. % coinductive failure unavoidable
check_chs_and_call_stack(X, success) :-
    exact_match_in_chs(X), !. % coinductive success unavoidable
check_chs_and_call_stack(X, Result) :- % avoid failure, if possible.
    constrain_goal_against_unifiable_duals(X), % non-deterministic.
    check_call_stack(X, Result).

% Check the call stack for cycles on current goal. If a cycle over an
% exact match is found, don't look for other matches when backtracking.
check_call_stack(X, failure) :-
    call_stack_has_positive_cycle_w_exact_match(X), !.
check_call_stack(X, success) :-
    call_stack_has_even_cycle_w_exact_match(X), !.
check_call_stack(X, success) :-
    call_stack_has_even_cycle(X), % non-deterministic
    unify_goal_w_match.
check_call_stack(X, expand).
```

Fig. 2. Abstracted s(ASP) Meta-interpreter

Theorem 2
Our predicate method is sound for legal programs grounded over the s(ASP) universe.

Proof
Our propositional method was originally proven sound with respect to the original GL method (Marple et al. 2012). It is a simple matter to extend the original soundness proof to show the equivalence of our propositional method to the modified GL method described in Section 3.1.1, thereby proving it sound for legal programs grounded over the s(ASP) universe. Thus, by Theorem 1, our predicate method is also sound for legal programs grounded over the s(ASP) universe.   □

4.2 Predicate Programs

The case of predicate logic programs is significantly more complicated than the propositional case. While s(ASP) is sound for all legal programs, completeness over this class is impossible, as we will discuss.

4.2.1 Soundness

Informally, the s(ASP) algorithm is sound if every partial model it generates is part of some stable model of the program. To show this we must first separate the (grounded) program into three parts:

- The set of clauses required to prove the partial model.
- The set of clauses related to the query but not needed to prove the partial model.
- The set of clauses not related to the query.

We will first show that the second set of clauses can be removed without affecting the result. That is, the partial model is a part of some stable model of the new program, and that stable model is a stable model of the original program. We also show that clauses needed to prove that some literal is false can be modified by removing goals as long as one "false" goal remains. Using these two modifications and the splitting theorem (Lifschitz and Turner 1994), we can isolate a subprogram comprised only of clauses and literals touched by the s(ASP) algorithm.

Using the terminology of (Lifschitz and Turner 1994), detailed in the next section, this subprogram can be considered the "bottom" of the program, with the remaining clauses forming the "top" of the program. We show that the partial model generated by the s(ASP) algorithm is a stable model of the bottom of the program. Then, in accordance with the splitting theorem we transform the top of the program and show that since the NMR check is satisfied a stable model exists for it. By using the splitting theorem and the points discussed above, we show that the union of the partial model and the top's stable model is a stable model the original program.

Before continuing, we define the following for convenience:

Definition 28
Let G be a goal constructed from atom A. Then, atom(G) = A, and goal(A) is
the set of both goals (positive and negative) that can be constructed from A. The
arguments of A are said to be the arguments of G.

Definition 29
Let R be a clause of the form:

   p :- q$_1$, ..., qi, ..., q$_m$,
        not r$_1$, ..., not r$_j$, ..., not r$_n$.

Then:

 • **head(R)** = p
 • **pos(R)** = { q$_1$, ..., qi, ..., q$_m$ }
 • **neg(R)** = { not r$_1$, ..., not r$_j$, ..., not r$_n$ }
 • **lit(R)** = { H } $\cup$ pos(R) $\cup$ neg(R)

Review of Splitting Theorem Next, we review the splitting theorem (Lifschitz and Turner 1994).
A set of literals can be used to *split* a ground program. This set is called a *splitting
set*, and is defined as follows.

Definition 30
A set of ground atoms U is a **splitting set** for some ground program P if for every
clause R in P, head(R) $\in$ U $\Rightarrow$ lit(R) $\subseteq$ U.

The program is divided into two parts, the top and the bottom. The bottom
is the set of clauses related to the splitting set and the top is the set of all other
clauses.

Definition 31
Let P be a ground program and U a splitting set for P. The **bottom** of P with
respect to U, specified as b$_U$(P), is the set of clauses r for which lit(r) $\subseteq$ U. The set
P $\setminus$ b$_U$(P) is the **top** of P with respect to U.

The stable models of a ground program P can be computed by combining the
stable models of b$_U$(P) and the stable models generated by the top. This requires
us to generate a new program from P $\setminus$ b$_U$(P) based on the stable models for b$_U$(P).

Theorem 3
Let P be some ground program, U a splitting set of P, and X a stable model for
b$_U$(P). For each clause r in P such that pos(r) $\subset$ X and neg(r) $\cap$ X = $\emptyset$, we define
a new clause r$'$ with:

 • head(r$'$) = head(r),
 • pos(r$'$) = pos(r) $\setminus$ U
 • neg(r$'$) = neg(r) $\setminus$ U

We define the program e$_U$(P $\setminus$ b$_U$(P), X) as the set of of all such new clauses,
and for some stable model Y of e$_U$(P $\setminus$ b$_U$(P), X), X $\cup$ Y is a stable model of P.

If either b$_U$(P) or e$_U$(P $\setminus$ b$_U$(P), X) has no stable model then there is no stable
model for P.

Proof
Proofs for these results are available in the original paper which introduced the
splitting theorem (Lifschitz and Turner 1994).   □

Stripping Unneeded Rules and Body Literals

Lemma 1
Let P be a ground program, and M a stable model of P. Let R be a clause not in P
such that the head of R is in M. Then, M is a stable model of the program P ∪ { R }.

Proof
There are two cases for R:

Case 1: There exists L ∈ neg(R) such that L ∈ M. In this case R is removed when
computing the reduct, and the reduct does not change. Therefore, M is the least
model of the reduct.

Case 2: R is not removed when the reduct is computed. We will call the trans-
formed clause R′.

Since the head of R is in M there must exist some clause R_2 in P with head(R) =
head(R_2) such that for all L ∈ pos(R_2), L ∈ M and for all L ∈ neg(R_2), L ∉ M. We
will call the transformed clause in the reduct R″.

The only way R′ can affect the least model of P ∪ { R } is to be used to place its
head in it. Therefore, since R′ can not affect literals besides its head, the body
literals in R″ must be in the least model of the reduct of P ∪ { R }. Thus, head(R″)
(which is also head(R′)) must also be in the least model, and the least model for
the reduct of P ∪ { R } is the same as for P's reduct. So, M must be the least
model of the reduct of P ∪ { R }.

Thus, M is a stable model of P ∪ { R }.
   □

Lemma 2
Let P be a ground program and M a stable model of P. Let R be a clause in P such
that there exists a literal in the body that is not in M, and let G be a ground goal.
Let P′ be the program constructed by adding G to the body of R. M is a stable model
of P′.

Proof
Let P be a ground program, and M be a stable model of P. Let R be a clause in P
such that the head of R is not in M. Let G be some ground goal.
   Create new program P′ by adding G to the body of R. Call this clause R′.
   We have two cases:

1. There exists some literal L ∈ neg(R) and L ∈ M, or
2. there is some literal L ∈ pos(R) such that L ∉ M.

In case 1, we know that R′ will be removed when computing the reduct, and thus can not affect the least model. So we only need to consider case 2. In addition we can assume that G does not cause the clause to be removed from the reduct(otherwise it can not affect the least model). Now, notice that the addition of G does not affect the truth value of L. Thus, it is possible that the same process that causes L to not be in the least model of the reduct of P will cause it to not be in the reduct of P′. So, R′ cannot be used to place its head in the least model. Since no other clauses have changed, the least model of the reduct of P′ is the same as the one for P, and thus M is a stable model of P′.   □

To prove theorem 4, we want to separate the part needed to prove the query from the rest. Since the s(ASP) algorithm is goal directed we need to remove clauses related to the query that are not needed to prove it and goals in clauses related to the query, but not needed to prove it. We call this process **trimming**. Then we will make use of the splitting theorem from (Lifschitz and Turner 1994) to divide the new program into two parts, treating the portion needed to prove the query as the bottom, and the rest as the top.

It is important to remember that the s(ASP) algorithm works directly with the ungrounded program, but we will be trimming its ground program. When executing a clause (using it to prove some goal) it is possible (and likely) that variables in the clause will be constrained or bound by some goals in its body. If we associate with a clause a function for the domains of the variables in it, we can treat the state of the clause before and after as separate clauses.

Definition 32
Let R be a clause.

Let $\delta(\text{R}, \text{X})$ be the domain (set of possible groundings) of the variable X in R. X may be bound or constrained (though the constraint list may be empty). A clause in a program before execution will always have all variables unconstrained and therefore they will have the s(ASP) universe as their domains.

The result of the execution of R succeeding is called $\sigma(\text{R})$. The variables in $\sigma(\text{R})$ are the same as in R, and for all variables X in R, $\delta(\sigma(\text{R}), \text{X}) \subseteq \delta(\text{R}, \text{X})$.

Definition 33
To **trim** a program we will follow the following algorithm. Let P be a program, P′ be the result of grounding P over the s(ASP) universe, M a partial model of P generated by the s(ASP) algorithm, and M′ the grounding of M over the s(ASP) universe. Let $\Phi(\text{R})$ be the set of clauses in P′ generated by grounding clause R in P.

1. While M is begin computed: Let G be the current goal such that G is not a negated goal or a built-in/system generated literal such as the NMR check. Let R be the clause that is selected. If the execution of R succeeds, then mark all clauses in P′ that are in $\Phi(\text{R})$ and could be considered a grounding of $\sigma(\text{R})$.
2. After M is computed: Create a new program P″ from P′ by:

   - removing all clauses R from P′ for which the head(R) $\notin$ M′ and R is not marked, and

- transform all clauses R in P′ for which not head(R) ∈ M′ by removing all body goals for which nether they nor their negations are in M.

3. P″ is the result of **trimming** P′ with respect to M′.

Forall

Lemma 3

Let P be a s(ASP) program, G be an atom, and X an unconstrained variable in G. Let $\mathcal{G}$ be the set of all goals obtained by grounding X in G over the s(ASP) universe. Then, a forall(X, G) in P succeeds if and only if all goals in $\mathcal{G}$ succeed.

Proof

Assume the opposite is true. That is, either forall(X, G) succeeds and some L ∈ $\mathcal{G}$ fails, or all goals in $\mathcal{G}$ succeed, but forall(X, G) fails.

Case 1: Suppose forall(X, G) succeeds, but there exists some L ∈ $\mathcal{G}$ such that L fails. There are two phases for the forall to succeed. First, G must succeed with X unbound. Then prove G with X grounded with each of its constraints. Since L fails, the value corresponding to X in L could not have been in the constraint list. Otherwise, the forall would have failed in the second phase. However, we could take the proof tree generated by the first phase, and ground X to obtain a proof tree for L, meaning there is a way for L to succeed. A contradiction.

Case 2: Suppose forall(X, G) fails, but all L ∈ $\mathcal{G}$ succeed. We know that the forall could not have failed in the second phase, otherwise there would be some L ∈ $\mathcal{G}$ such that L fails. Therefore there are two possibilities. Either, there is no way for G to succeed or all ways require X to be ground. The second case cannot be the case since all L ∈ $\mathcal{G}$ succeed, and by definition, the s(ASP) universe contains an infinite number of terms that do not appear in the herbrand universe of P. Thus there exists some term in the s(ASP) universe for which X cannot be explicitly grounded against. The first case also cannot be true since all L ∈ $\mathcal{G}$ succeed, thus there must be a way for G to succeed. A contradiction.

Therefore, forall(X, G) succeeds if and only if all L ∈ $\mathcal{G}$ succeeds. □

Constructive Coinductive Failure

Lemma 4

Let P be a program, G a goal currently in the CHS, and G′ be a goal that unifies with the negation of G that we wish to prove. Let G″ be the goal generated by the constructive coinductive failure algorithm from G and G′. Then, G does not unify with G″.

Proof

Without loss of generality, assume G is negated. So, G′ will not be.

If we assume that given an argument in G′ we can restrict it so that it does not unify with the corresponding argument in G, we can easily see that if the algorithm

succeeds G″ cannot unify with atom(G). So, we must show that if we restrict an argument (the algorithm succeeds for the argument) in G′ it is never the case that it unifies with the corresponding argument in G.

We will show this by inducting over the depth of the term. The depth of a variable or constant is zero, and the depth of a list or function is one more than the maximal depth of all its arguments. Let T′ be the argument from G′ and T the corresponding argument from G. Notice that T and T′ must unify. If T is a loop variable, then T′ will be added to its constraint list, even if T′ is a variable. In this case it is obvious that T and T′ no longer unify. If T is an unconstrained variable and not a loop variable, the algorithm will always fail(since it unifies with everything), so we will not explicitly consider this case below.

Base Case: Suppose T′ has a depth of zero. If it is a constant the algorithm fails, so we only need to consider the case it is a variable. The behavior of the algorithm depends on what T is.

- If T is a constant, then T′ is constrained against it. It is obvious in this case that they do not unify.
- If T is a constrained variable, then a term is nondeterministically selected from its constraint list for which T′ is not constrained against, and is used to ground T′. Again, it is obvious they do not unify.
- If T is a list or function, T will be added to the constraint list for T′, and therefore no longer unify with T′.

Inductive Hypothesis: Let T'_2 and T_2 be terms that unify, with T'_2 having a depth less than or equal to k and T_2 being from the goal in the CHS. Assume that if the algorithm succeeds the goal generated will not unify with T_2.

Inductive Step: Suppose T′ has a depth of k + 1. Then, T′ must be a list or function, and since T unifies with T′ it must also be a list or a function with the same functor and arity. So, we nondeterministically select an argument in T′ and apply the algorithm to it with the corresponding argument in T. That argument must have depth of k or less, and by the inductive hypothesis if the algorithm succeeds then it cannot unify with the corresponding argument in T, and therefore by replacing the argument in T′ with the result, we know that the new term cannot unify with T. If no argument can succeed then the algorithm will fail, so such a case can be ignored.

Thus, by induction if the algorithm succeeds the resulting term will not unify with the term in the CHS, and therefore G″ will never unify with atom(G). □

Soundness Theorem

Theorem 4
Let P be a program, and M a partial model of P generated by the s(ASP) algorithm. Let M_2 and P_2 be the results of grounding M and P, respectively, over the s(ASP) universe. There exists a stable model X of P_2 such that for all literals L in M_2, L is in X, and for all literals L with not L in M_2, L is not in X.

Proof of Theorem 4

Let `P` be a program, and `M` a partial model of `P` generated by the s(ASP) algorithm. Let M_2 and P_2 be the results of grounding `M` and `P`, respectively, over the s(ASP) universe. If `M` contains loop variables then we may choose a domain for each loop variable that does not contradict the rest of `M` without loss of generality. This is just selecting one out of the infinite number of partial models represented by `M`. Assume that there exists at least one assignment that contains no empty variables since we consider such a situation as a failure.

Before proving our claim we must show that M_2 is consistent. That is, for some literal `L` it is not the case that `L` and `not L` are both in M_2. First, notice that if the value of `L` depends on `not L` (and visa versa) then the s(ASP) algorithm will fail or the goal will be constrained so that `L` and `not L` will not be in the grounding. This is because it is an odd cycle over negation. So we only need to consider the case where `G` $\in$ goal(`L`) unifies with something in `M`, but we want to prove a goal that unifies with the negation of `G`. However, by lemma 4 we know that the second goal will be restricted so that it no longer unifies with `L` or `not L`. So, M_2 is consistent.

Let P_3 be the result of trimming P_2 with respect to M_2, and `S` be a set of literals such that `L` $\in$ `S` $\iff$ `L` $\in M_2 \lor$ `not L` $\in M_2$. `S` is a splitting set of P_3. Now we must do two things. First we must show that M_2 is a stable model of the bottom, and that there exists a stable model for the top.

To prove that M_2 is a stable model of the bottom we must show that:

1. All literals in M_2 are in the least model of the reduct for P_3, and
2. No literal `L` with `not L` in M_2 will be in it.

   Case 1: For all literals `L` $\in M_2$: Let `L'` be the non-ground atom in `M` that is used to generate `L`, and `R` be the clause in `P` that is used to prove `L'`. We can construct a tree by using `L'` as the root, and the body literals from `R` as the children. The negated goals will be in `M` and later removed from the clause when computing the reduct. So, we can ignore them and only consider literals as children. Additionally, we will keep the groundings and constraints of the variables at the time of success. Then ground the tree such that the resulting tree has `L` as the root. This corresponds to a clause in P_3, since it would have been marked and therefore not removed. The leaves of such a tree must have facts in the reduct of P_3, and therefore will be in the least model. From there we know that the root of each level going up the tree will be in the least model, including `L`.

   Case 2: For all literals `L` such that `not L` $\in M_2$ we must show that there is no way `L` can be in the least model of the reduct for P_3. Firstly, if a clause with `L` as the head is part of a positive cycle for `L`, then it cannot be used to put `L` into the least model. So, we only need to consider noncyclic cases. For these cases we will prove it inductively, and to do that we will define the **level** of a clause. If a clause is a fact then it has level zero. For non-fact clauses, we say that a body goal `B` has a level equal to that of the highest level clause with atom(`B`) as the head. The level of all non-facts is one plus the highest level of the body literals. In the case of a body literal for which there are no clauses, it is considered level zero.

Base Case: Let L be a literal such that $not \; L \in M_2$. There cannot be a fact for L, otherwise $not \; L$ could not be in M_2. The goal $not \; L$ comes from the success of a dual rule, which would always fail if a fact for L existed. If L has no clauses, then it cannot be in the least model.

Inductive Hypothesis: Let L be a literal such that $not \; L \in M_2$. Suppose all clauses with a level less than or equal to k cannot be used to place L in the least model.

Inductive Step: Let L be a literal such that $not \; L \in M_2$. Let R be a clause with L in the head and a level of $k + 1$. In order for the dual to succeed and allow $not \; L$ to be in the grounding there must be a goal G in R such that $G \notin M_2$. Since $G \notin M_2$ but was not trimmed from R the negation of G must be in M_2. If G is negated then R would be removed when computing the reduct. So, we only need to consider the case G is not negated. G must have a level of at most k, and by the inductive hypothesis we know that there is no way to place G into the least model of the reduct, and therefore R cannot be used to place L into the least model.

Thus, by induction L is not in the least model of the reduct for P_3.

Therefore, M_2 is a stable model of the bottom of P_3 with respect to the splitting set S.

Now we must show that there exists a stable model for the top. We will do this by observing that the only way for there not to be a stable model is if there is an inconsistency, and there can be an inconsistency only if there is an odd cycle. So, we will show that the modified program from the top will contain no odd cycles.

Let R be an OLON in P_2, and R' the clause in P such that when grounding R' over the s(ASP) universe, R is generated. Since R is part of an odd cycle, R' is also considered part of an odd cycle since we only look at predicate name and arity. Thus there will be a NMR check for R'. By lemma 3, we can treat the NMR check as a conjunction of checks with the head grounded over the s(ASP) universe. So, either there exists a body literal in R with its negation in M_2 or the head of R is in M_2. In the second case, R will either be removed through trimming or will be in the bottom of P_3. For the first case, assume R is not removed through trimming or in the bottom of P_3. Then R will be removed when computing the partial evaluation for the top since the negation of some literal in the body is in M_2. Thus, there are no odd cycles when computing the answer sets of the top.

It is apparent from the splitting theorem that if $L \in M_2$ is a literal then L is in X. So, we only need to show that if $not \; L \in M_2$ then $L \notin X$. First, notice that the truth value of L is determined by the bottom of P_3 with respect to S, and cannot be in the stable model of the top. Thus, L cannot be in X.

By lemma 1, we know that a stable model for P_3 is also a stable model of P_2. $\square$

4.2.2 Completeness

While s(ASP) is sound for the set of all legal programs, completeness for this set is impossible. While we have striven to make s(ASP) complete for the largest class of

programs possible, we leave the precise definition of this class and the associated proofs to future work.[1] Instead, we argue that the utility of our method outweighs its lack of completeness.

It is easy enough to show that s(ASP) cannot possibly be complete for all legal programs. Consider the class of *stratified programs*, that is, programs with no loops over negation. A stratified program will always have a unique stable model which coincides with its *perfect model*, the model produced by the perfect model semantics (Cadoli and Schaerf 1993). However, the perfect model of such a program may be incomputable (Apt and Blair 1990). Therefore, even though a stable model must exist for such a program, we may be unable to compute it. As such, it is not possible to guarantee completeness for such programs.

The trade-off for this loss of completeness is a massive increase in expressive power. The propositional stable model semantics can only express relations which are co-NP, however, the predicate stable model semantics and s(ASP) can express relations which are Π_1^1 (Schlipf 1990; Cadoli and Schaerf 1993).

In addition to increased computational expressiveness, s(ASP) supports lists, complex data structures and real numbers, providing programmers with tools not found in any other implementation of the stable model semantics. With these features, even programs which can already be expressed in the propositional semantics may be easier to write in s(ASP).

Aside from completeness itself, the only "desirable" property that we lose compared to other implementations of the stable model semantics is the guarantee that a program will always terminate. However, guaranteed termination is a double-edged sword. It implies that only decidable problems can be encoded, as, by definition, this guarantee cannot be applied to semi-decidable or undecidable problems. By abandoning guaranteed termination, we are able to support programs which encode such problems, something that no other implementation of the stable model semantics can claim. This is significant, as problems which are undecidable in general may still produce useful results for some cases.

Thus, we trade completeness for superior functionality and the ability to encode problems which no other implementation of the stable model semantics can handle. While completeness is certainly desirable, it is our firm belief that the gains derived from this trade significantly outweigh the losses.

5 Implementation and Examples

A fully functional prototype implementation of the method presented here has been created, also using the name s(ASP). The implementation is written in Prolog and totals about 4,300 lines of code (excluding comments and blank lines). An open source release is available at (Marple 2015).

Unlike its predecessor, Galliwasp, s(ASP) is completely self-contained: neither a grounder nor a separate compiler is required. As with our method, the prototype

---

[1] The exception being finite, legal, grounded programs, for which we have proven completeness in Section 4.1.

will accept any legal normal logic program and execute it *without grounding any portion of the program at any stage.* While the prototype is not designed to be competitive in terms of speed, our method allows it to offer features not found in any other implementation of the stable model semantics, including answer set programming systems. In the following subsections, we will look at how s(ASP) behaves with a number of examples.

5.1 Example: N Queens with Lists

A variant of the N queens problem using lists, can be found in Figure 3. This example is of particular interest, as it has no finite grounding and thus cannot be run by other implementations of the stable model semantics. Additionally, the even loop in the last two lines of the code will produce two loop variables, discussed in Section 3.2.2.

When executed by our prototype implementation the user will get the following:

```
?- nqueens(5,X).
{ nqueens(5,[q(1,2),q(2,4),q(3,1),q(4,3),q(5,5)]), q(1,2), q(2,4),
q(3,1), q(4,3), q(5,5) }
X = [q(1,2),q(2,4),q(3,1),q(4,3),q(5,5)].

?- nqueens(4,X).
{ nqueens(4,[q(1,2),q(2,4),q(3,1),q(4,3)]), q(1,2), q(2,4), q(3,1),
q(4,3) }
X = [q(1,2),q(2,4),q(3,1),q(4,3)];
{ nqueens(4,[q(1,3),q(2,1),q(3,4),q(4,2)]), q(1,3), q(2,1), q(3,4),
q(4,2) }
X = [q(1,3),q(2,1),q(3,4),q(4,2)];
false.
```

As no grounding is performed, multiple instances of the problem can be queried in a single session. The sample output illustrates this by querying both four and five queens. While the entire partial stable model is provided, variables in the query are printed, making desired information much easier to find. In the above example, `X` will be bound to the list of queens selected. As with Prolog interpreters, ';' and '.' can be used to reject or accept a solution, respectively. As there are only two solutions for four queens, pressing ';' a second time leads to failure. Note that the output will often contain variables with names consisting of "Var" followed by an integer. This is simply because s(ASP) renames variables to ensure that they are unique.

5.2 Example: Hamiltonian Cycle Detection

Figure 4 contains an encoding of the Hamiltonian cycle problem, along with a simple graph. The results of this example provide an interesting look at our use of negatively constrained variables in output. While another solution exists, due

```
% solve the N queens problem for a given N, returning a list of queens as Q
nqueens(N, Q) :-
    nqueens(N, N, [], Q).

% pick queens one at a time and test against all previous queens
nqueens(X, N, Qi, Qo) :-
    X > 0,
    pickqueen(X, Y, N),
    not attack(X, Y, Qi),
    X1 is X - 1,
    nqueens(X1, N, [q(X, Y) | Qi], Qo).
nqueens(0, _, Q, Q).

% pick a queen for row X.
pickqueen(X, Y, Y) :-
    Y > 0,
    q(X, Y).
pickqueen(X, Y, N) :-
    N > 1,
    N1 is N - 1,
    pickqueen(X, Y, N1).

% check if a queen can attack any previously selected queen
attack(X, _, [q(X, _) | _]). % same row
attack(_, Y, [q(_, Y) | _]). % same col
attack(X, Y, [q(X2, Y2) | _]) :- % same diagonal
    Xd is X2 - X, abs(Xd, Xd2),
    Yd is Y2 - Y, abs(Yd, Yd2),
    Xd2 = Yd2.
attack(X, Y, [_ | T]) :-
    attack(X, Y, T).

q(X, Y) :- not negq(X, Y).
negq(X, Y) :- not q(X, Y).

abs(X, X) :- X >= 0.
abs(X, Y) :- X < 0, Y is X * -1.
```

Fig. 3. N Queens Program with Lists

to space limitations, only the first is provided. The cycle is represented by the
chosen/2 elements at the beginning of the set:

```
?- reachable(0).
{ chosen(0,1), chosen(1,2), chosen(2,3), chosen(3,4), chosen(4,0),
edge(0,1), edge(1,2), edge(2,3), edge(3,4), edge(4,0), edge(4,1),
edge(4,2), edge(4,3), other(0,0), other(0,2), other(0,3),
other(0,4), other(1,0), other(1,1), other(1,3), other(1,4),
other(2,0), other(2,1), other(2,2), other(2,4), other(3,0),
```

```
reachable(V) :- chosen(U, V), reachable(U).
reachable(0) :- chosen(V, 0).

% Every vertex must be reachable.
:- vertex(U), not reachable(U).

% Choose exactly one edge from each vertex.
other(U, V) :-
    vertex(U), vertex(V), vertex(W),
    V \= W, chosen(U, W).
chosen(U, V) :-
    vertex(U), vertex(V),
    edge(U, V), not other(U, V).

% Two edges cannot be incident on the same
% vertex.
:- chosen(U, W), chosen(V, W), U \= V.

% Sample graph: vertexes and the edges connecting them.
vertex(0).
vertex(1).
vertex(2).
vertex(3).
vertex(4).

edge(0, 1).
edge(1, 2).
edge(2, 3).
edge(3, 4).
edge(4, 0).
edge(4, 1).
edge(4, 2).
edge(4, 3).
```

Fig. 4. A program for Hamiltonian cycle detection with a simple graph included.

```
other(3,1), other(3,2), other(3,3), other(4,1), other(4,2),
other(4,3), other(4,4), reachable(0), reachable(1), reachable(2),
reachable(3), reachable(4), vertex(0), vertex(1), vertex(2),
vertex(3), vertex(4), not chosen(0,0), not chosen(0,2), not
chosen(0,3), not chosen(0,4), not chosen(0,Var644) ( Var644 \= 0,
Var644 \= 1, Var644 \= 2, Var644 \= 3, Var644 \= 4 ), not
chosen(1,0), not chosen(1,1), not chosen(1,3), not chosen(1,4), not
chosen(1,Var710) ( Var710 \= 0, Var710 \= 1, Var710 \= 2, Var710 \=
3, Var710 \= 4 ), not chosen(2,0), not chosen(2,1), not chosen(2,2),
not chosen(2,4), not chosen(2,Var776) ( Var776 \= 0, Var776 \= 1,
Var776\= 2, Var776 \= 3, Var776 \= 4 ), not chosen(3,0), not
chosen(3,1), not chosen(3,2), not chosen(3,3), not chosen(3,Var842)
```

```
( Var842 \= 0, Var842 \= 1, Var842 \= 2, Var842 \= 3, Var842 \= 4 ),
not chosen(4,1), not chosen(4,2), not chosen(4,3), not chosen(4,4),
not chosen(4,Var908) ( Var908 \= 0, Var908 \= 1, Var908 \= 2, Var908
\= 3, Var908 \= 4 ), not chosen(Var627,_) ( Var627 \= 0, Var627 \=
1, Var627 \= 2, Var627 \= 3, Var627 \= 4 ), not chosen(Var663,1) (
Var663 \= 0, Var663 \= 1, Var663 \= 2, Var663 \= 3, Var663 \= 4 ),
not chosen(Var734,2) ( Var734 \= 0, Var734 \= 1, Var734 \= 2, Var734
\= 3, Var734 \= 4 ), not chosen(Var805,3) ( Var805 \= 0, Var805 \=
1, Var805 \= 2, Var805 \= 3, Var805 \= 4 ), not chosen(Var876,4) (
Var876 \= 0, Var876 \= 1, Var876 \= 2, Var876 \= 3, Var876 \= 4 ),
not chosen(Var922,0) ( Var922 \= 0, Var922 \= 1, Var922 \= 2, Var922
\= 3, Var922 \= 4 ), not edge(0,0), not edge(0,2), not edge(0,3),
not edge(0,4), not edge(1,0), not edge(1,1), not edge(1,3), not
edge(1,4), not edge(2,0), not edge(2,1), not edge(2,2), not
edge(2,4), not edge(3,0), not edge(3,1), not edge(3,2), not
edge(3,3), not edge(4,4), not other(0,1), not other(1,2), not
other(2,3), not other(3,4), not other(4,0), not vertex(Var31) (
Var31 \= 0, Var31 \= 1, Var31 \= 2, Var31 \= 3, Var31 \= 4 ) } .
```

6 Applications

The s(ASP) system is publicly available (Marple 2015), and has been used to develop a number of non-trivial applications based on ASP; it has also been used to organize an AI hackathon (UT Dallas AI Society 2016). Some of these applications cannot be executed on traditional ASP systems such as CLASP, as these applications make use of lists and structures to represent information. They have been developed by people who are not experts in ASP. These applications include:

1. **Degree Audit System:** A system for automatically performing a degree audit of a student's undergraduate transcript at a US University, i.e., automatically determining whether a student can graduate with a degree or not, has been developed using the s(ASP) system (Sobhi and Srirangapalli 2016). The system represents the graduation requirements laid out in the course catalog as ASP clauses. Use of negation is important for representing these requirements. The system has to make use of lists, and has hundreds of courses that appear as constants in the program (hence its grounding will produce an inordinately large program).

2. **Physician Advisory System:** A system for disease management, particularly, for chronic heart failure has been developed using the s(ASP) system (Chen et al. 2016). This system automates the 80-page guidelines (that the American College of Cardiology has developed) by representing them in ASP. While the current system can be run under systems such as CLASP due to the number of constants not being too large, the final system that models a doctor's full knowledge will have quite a few constants, and advanced data-structures may be needed.

3. **Automating Textbook Knowledge:** A system that represents high-school level knowledge about cells (in the discipline of biology) as answer set programs has been developed using s(ASP). It can answer high-school level questions posed as s(ASP) queries. The goal is to represent the knowledge in the entire introductory biology textbook as an answer set program, and then be able to automatically answer questions that would be asked of a student (the questions have to be translated into ASP queries that are then executed to find the answer).

4. **Birthday Gift Advisor:** A recommendation system for birthday gifts has also been developed using the s(ASP) system. This system codes a person's knowledge about friends, level of friendship, a person's wealth level, generosity level, and hobbies as answer set programs. When queried, the system can recommend a birthday present for a particular friend (e.g., on one's Facebook page). Note that other similar recommendation systems can also be built using s(ASP).

7 Related and Future Work

With respect to related work, most of it focuses on answer set programming rather than purely on stable models. Perhaps most notably, DLV supports lists and structures, however, the underlying execution mechanism is still based on grounding and then finding the stable models of the resulting propositional program (Bihlmeyer et al. 2017; Maratea et al. 2008; Leone et al. 2006). To ensure that the grounded program stays finite, the DLV system resorts to techniques such as *finite domain checking* and requiring that the programmer specify an *upper integer limit* (Bihlmeyer et al. 2017), i.e., the maximum numerical value allowed in the program. In contrast, no finite domain checks or upper integer limits are required by the s(ASP) system.

Various other attempts have been made to achieve a goal-directed method for executing propositional answer set programs (Alferes et al. 2004; Bonatti et al. 2008; Bonatti 2001; Pereira and Pinto 2005; Pereira and Pinto 2009; Shen et al. 2004). However, all of these either alter the semantics or significantly restrict the programs accepted, and all are restricted to grounded programs. Our own method will accept any normal logic program without altering the underlying stable model semantics. Similarly, work has been done in the area of predicate answer set programming, but only with much more severe restrictions on accepted programs (Bonatti 2004; Heymans and Vermeir 2003). Other ASP systems ground "on the fly", but grounding is still performed at some point during execution (Dal Palù et al. 2009; Dao-Tran et al. 2012; Lefvre and Nicolas 2009a; Lefvre and Nicolas 2009b). A variety of efforts have focused on constructive negation (Chan 1988; Stuckey 1991; Pearce and Valverde 2005), but our method's combination of negatively constrained variables and specially adapted dual rules represents a unique approach.

There are a vast number of potential routes for future work, including practical applications and extensions to the language. In particular, the restriction that left recursion cannot lead to success should be resolvable through the addition of tabling (Swift and Warren 2012). We also plan to invest effort on improving our im-

plementation's efficiency. This will be achieved by designing a WAM-style abstract machine specialized for our method and developing an emulator for it. Normal logic programs can then be compiled to this abstract machine and executed using the emulator, much in the style of modern Prolog systems.

8 Conclusion

In this paper, we have presented a method for computing partial stable models of predicate normal logic programs and proven it sound for a large class of programs. The key to this method is the use of a special, non-Herbrand universe which allows us to ensure soundness while still producing useful results for a large class of programs. Our method also relies on coinduction and constructive negation to execute programs in a top-down manner similar to that used in Prolog systems. Compared to similar attempts, our method supports a much larger class of programs. Indeed, only three restrictions are placed on input programs: that arithmetic operations must be ground when executed, that left recursion cannot lead to success, and that two negatively constrained variables cannot be disunified with each other. An implementation of our method, s(ASP), is freely available (Marple 2015) and has already been used in the development of non-trivial applications.

Acknowledgment

References

ALFERES, J. J., PEREIRA, L. M., AND SWIFT, T. 2004. Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. *Theory and Practice of Logic Programming 4*, 383–428.

APT, K. R. AND BLAIR, H. A. 1990. Arithmetic Classification of Perfect Models of Stratified Programs. *Fundamenta Informaticae 13,* 1, 1–17.

BANSAL, A., SAEEDLOEI, N., AND GUPTA, G. 2010. Timed Planning. In *Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference.* AAAI Press.

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.

BARAL, C. AND GELFOND, M. 1994. Logic Programming and Knowledge Representation. *The Journal of Logic Programming 19*, 73–148.

BIHLMEYER, R., FABER, W., IELPA, G., LIO, V., AND PFEIFER, G. 2017. DLV - User Manual. `http://www.dlvsystem.com/html/DLV_User_Manual.html`.

BONATTI, P. A. 2001. Resolution for Skeptical Stable Model Semantics. *Journal of Automated Reasoning 27*, 391–421.

BONATTI, P. A. 2004. Reasoning with Infinite Stable Models. *Artificial Intelligence 156,* 1, 75–111.

BONATTI, P. A., PONTELLI, E., AND SON, T. C. 2008. Credulous Resolution for Answer Set Programming. In *Proceedings of the 23rd national conference on Artificial Intelligence - Volume 1.* AAAI'08. AAAI Press, 418–423.

CADOLI, M. AND SCHAERF, M. 1993. A Survey of Complexity Results for Non-Monotonic Logics. *The Journal of Logic Programming 17,* 2-4, 127–160.

CHAN, D. 1988. Constructive Negation Based on the Completed Database. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium.* MIT Press, 111–125.

CHEN, W., SWIFT, T., AND WARREN, D. S. 1995. Efficient Top-Down Computation of Queries Under the Well-Founded Semantics. *J. Log. Program. 24,* 3, 161–199.

CHEN, Z., **Kyle Marple**, SALAZAR, E., GUPTA, G., AND TAMIL, L. 2016. A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns. *Theory and Practice of Logic Programming 16,* 5-6, 604–618.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae 96,* 3, 297–322.

DAO-TRAN, M., EITER, T., FINK, M., WEIDINGER, G., AND WEINZIERL, A. 2012. OMiGA: An Open Minded Grounding On-The-Fly Answer Set Solver. In *Logics in Artificial Intelligence.* Lecture Notes in Computer Science, vol. 7519. Springer Berlin Heidelberg, 480–483.

FITTING, M. 1985. A Kripke-Kleene Semantics for Logic Programs. *J. Log. Program. 2,* 4, 295–312.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Clasp: A Conflict-Driven Answer Set Solver. In *Proceedings of the 9th international conference on Logic Programming and Nonmonotonic Reasoning.* LPNMR'07. Springer-Verlag, 260–265.

GELFOND, M. AND LIFSCHITZ, V. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the Fifth international conference on Logic Programming.* MIT Press, 1070–1080.

GUPTA, G., BANSAL, A., MIN, R., SIMON, L., AND MALLYA, A. 2007. Coinductive Logic Programming and Its Applications. In *Proceedings of the 23rd international conference on Logic Programming.* ICLP'07. Springer-Verlag, 27–44.

HEYMANS, S. AND VERMEIR, D. 2003. Integrating Semantic Web Reasoning and Answer Set Programming. In *Proceedings of the 2nd International ASP Workshop.* ASP'03. CEUR-WS.org, 194–208.

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* POPL'87. ACM, 111–119.

LEFVRE, C. AND NICOLAS, P. 2009a. A First Order Forward Chaining Approach for Answer Set Computing. In *Logic Programming and Nonmonotonic Reasoning*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer Berlin Heidelberg, 196–208.

LEFVRE, C. AND NICOLAS, P. 2009b. The First Version of a New ASP Solver: ASPeRiX. In *Logic Programming and Nonmonotonic Reasoning*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer Berlin Heidelberg, 522–527.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic 7,* 3 (July), 499–562.

LIFSCHITZ, V. AND TURNER, H. 1994. Splitting a Logic Program. In *Proceedings of the Eleventh International Conference on Logic Programming.* ICLP'94. MIT Press, 23–37.

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artif. Intell. 157,* 1-2, 115–137.

LLOYD, J. 1987. *Foundations of Logic Programming.* Symbolic Computation: Artificial Intelligence. Springer-Verlag.

MARATEA, M., RICCA, F., FABER, W., AND LEONE, N. 2008. Look-back techniques and heuristics in DLV: implementation, evaluation, and comparison to QBF solvers. *J. Algorithms 63,* 1-3, 70–89.

MARPLE, K. 2014a. Design and Implementation of a Goal-directed Answer Set Programming System. Ph.D. thesis, University of Texas at Dallas.

MARPLE, K. 2014b. Galliwasp. `http://galliwasp.sourceforge.net`.

MARPLE, K. 2015. s(ASP). `https://sourceforge.net/projects/sasp-system/`.

MARPLE, K., BANSAL, A., MIN, R., AND GUPTA, G. 2012. Goal-directed Execution of Answer Set Programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. PPDP '12. ACM, New York, NY, USA, 35–44.

MARPLE, K. AND GUPTA, G. 2013. Galliwasp: A Goal-Directed Answer Set Solver. In *Logic-Based Program Synthesis and Transformation*. Lecture Notes in Computer Science, vol. 7844. Springer Berlin Heidelberg, 122–136.

MARPLE, K. AND GUPTA, G. 2014. Dynamic Consistency Checking in Goal-Directed Answer Set Programming. *Theory and Practice of Logic Programming 14,* 4–5, 415–427.

PEARCE, D. AND VALVERDE, A. 2005. A Order Nonmonotonic Extension of Constructive Logic. *Studia Logica 80,* 2–3, 321–346.

PEREIRA, L. AND PINTO, A. 2005. Revised Stable Models - A Semantics for Logic Programs. In *Progress in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 3808. Springer-Verlag, 29–42.

PEREIRA, L. AND PINTO, A. 2009. Layered Models Top-Down Querying of Normal Logic Programs. In *Practical Aspects of Declarative Languages*. Lecture Notes in Computer Science, vol. 5418. Springer-Verlag, 254–268.

SCHLIPF, J. S. 1990. The Expressive Powers of the Logic Programming Semantics. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 196–204.

SHEN, Y., YOU, J., AND YUAN, L. 2004. Enhancing Global SLS-Resolution with Loop Cutting and Tabling Mechanisms. *Theoretical Computer Science 328,* 3, 271–287.

SHOENFIELD, J. R. 1967. *Mathematical logic*. AK Peters Series. Association for Symbolic Logic.

SOBHI, A. AND SRIRANGAPALLI, S. 2016. Graduation Audit System. `https://gitlab.com/saikiran1096/gradaudit/`.

STUCKEY, P. 1991. Constructive Negation for Constraint Logic Programming. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*. LICS'91. IEEE Computer Society, 328–339.

SWIFT, T. S. AND WARREN, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *TPLP 12,* 1-2, 157–187.

ULLMAN, J. 1994. Assigning an Appropriate Meaning to Database Logic with Negation. In *Computers as Our Better Partners*. World Scientific Press, 216–225.

UT DALLAS AI SOCIETY. 2016. s(ASP) Hackathon. `https://hackai16.devpost.com/submissions`.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM 38,* 3, 620–650.