Abduction in Logic Programming: A New Definition and an Abductive Procedure Based on Rewriting*

Fangzhen Lin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8

Abstract

A long outstanding problem for abduction in logic programming has been on how minimality might be defined. Without minimality, an abductive procedure is often required to generate exponentially many subsumed explanations for a given observation. In this paper, we propose a new definition of abduction in logic programming where the set of minimal explanations can be viewed as a succinct representation of the set of all explanations. We then propose an abductive procedure where the problem of generating explanations is formalized as rewriting with confluent and terminating rewrite systems. We show that these rewrite systems are sound and complete under the partial stable model semantics, and sound and complete under the answer set semantics when the underlying program is so-called odd-loop free. We discuss an application of abduction in logic programming to a problem in reasoning about actions and provide some experimental results.

Key Words. Abduction, answer set programming, rewrite systems.

*This paper is extended from the short version that appeared in the proceedings of IJCAI 2001.

1 Introduction

Abductive reasoning subscribes to reasoning processes where explanatory hypotheses are formed and evaluated with respect to a knowledge base and an observation. Many intelligent tasks, including medical diagnosis, fault diagnosis, scientific discovery, legal reasoning, and natural language understanding, have been characterized as abduction.

In its most general form, the problem of abduction is this: given a background theory T and an observation q to explain, find an explanation theory Π such that $\Pi \cup T \models q$. Normally, we also want to put some additional conditions on Π, such as that it is consistent with T and contains only those propositions called abducibles. For instance, in propositional logic, given a background theory T, a set A of assumptions or abducibles, and a proposition q, an explanation S of q is commonly defined (see [18, 28, 30]) to be a minimal set of literals over A such that $T \cup S \models q$ and $T \cup S$ is consistent.

Abductive reasoning may be carried out in non-classic logics as well. Logic programming with stable models or answer sets as the underlying semantics has been considered particularly appealing for abduction, due to its applications in solving constraint satisfaction and other combinatorial problems, in expressing the frame axioms, in reasoning with actions and causality, and in representing the history of a plan [21, 26, 27].

In the context of logic programming, abduction has been investigated from both proof-theoretic and model-theoretic perspectives (e.g. [7, 14, 15, 16, 34]). One of the most followed definitions of abduction in logic programming is that of Kakas and Mancarella's generalized stable model semantics [15]. Given a logic program P, a set A of atoms standing for abducibles, and a query q, Kakas and Mancarella defined an abductive explanation S to be a subset of A such that there is an answer set (also called a stable model) of $P \cup S$ that satisfies q.

One can see the following two differences between this definition and the one that we defined above for propositional logic: In propositional logic, S is a set of literals, but in logic programming, it is just a set of atoms; In propositional logic, S must be minimal, in terms of the subset ordering relation; but there is no such requirement in the case of logic programming.

One could argue that these differences are due to the fact that under the answer set semantics, negation is considered to be "negation-as-failure." If none of the atoms in A appear in the head of a rule in the logic program P, then adding a set $S \subseteq A$ to P really means that we are adding the complete literal set, $S \cup \{\neg p \mid p \in A, p \notin S\}$, to P. This would also explain why there is no minimality condition in the definition: two complete sets of literals are never comparable in terms of the subset relation.

However, while this notion of abductive explanations makes sense in theory, it is problematic in practice. For instance, if $A = \{a, b\}$, and $P = \{q \leftarrow a.\ r \leftarrow b.\}$, then there are two abductive explanations for q according to Kakas and Mancarella's definition: $\{a\}$ and $\{a, b\}$. In general, if A has n elements, then there are 2^{n-1} abductive explanations for q, and in these exponentially many explanations, only a is relevant.

Since in this case a is the explanation that we are looking for, it is tempting here to say that we should prefer minimal abductive explanations like what we did for propositional logic. As we mentioned above, this does not make sense if we take an abductive explanation to be a complete set of literals as implied by the answer set semantics. However, one can still try to minimize the set of atoms, in this case, preferring $\{a\}$ over $\{a, b\}$.

However, this minimization strategy is problematic when a program contains negation. Consider a situation in which a boat can be used to cross a river if it is not leaking or, if it is leaking, there is a bucket available to scoop the water out of the boat. This can be axiomatized by the following logic program P:

$$canCross \leftarrow boat, \textsf{not}\ leaking.$$
$$canCross \leftarrow boat, leaking, hasBucket.$$

Now suppose that we saw someone crossed the river, how do we explain that? Clearly, there are two possible explanations: either the boat is not leaking or the person has a bucket with her. In terms of Kakas and Mancarella's definition, there are three abductive explanations for $canCross$, $\{boat\}$, $\{boat, hasBucket\}$, and $\{boat, leaking, hasBucket\}$, assuming that $A = \{boat, leaking, hasBucket\}$ is the set of abducibles. But only one of them, $\{boat\}$, is minimal.

On a closer look, we see that in our first example, when we say that $\{a\}$ is a preferred explanation over all the others, we do not mean the complete set of literals $\{a, \neg b\}$, is preferred over all the others. While we want a to be part of the explanation, we don't necessarily want $\neg b$ because we do not want to apply negation as failure on abducibles, which are assumptions one can make one way or the other. What we want is for the set $\{a\}$ itself to be the best explanation for q.

One way to justify this is that all possible ways of completing this set into a complete set of literals, $\{a, \neg b\}$ and $\{a, b\}$, turn out to correspond to all the abductive explanations of q according to Kakas and Mancarella's definition. The same kind of justification turns out to work for our second example as well: the reason that $\{boat\}$ is not a preferred explanation is that while its completion according to negation-as-failure, $\{boat, \neg leaking, \neg hasBucket\}$

is an explanation, some of its other completions, for example $\{boat, leaking, \neg hasBucket\}$ is not an explanation. This simple observation that for a set of literals to be an explanation, all of its possible extensions must also be an explanation will be the basis for our new definition of abduction in logic programming, given in Section 3.

With a new definition of abductive explanations in hands, we next address the computational problem of how to generate these explanations. To simplify our presentation, in Section 4, we shall consider first the special case when the set of abducibles is empty. In this case, abduction becomes query answering. Briefly speaking, given a logic program, we introduce a rewrite system consisting of its Clark completion as rewrite rules for literal rewriting, and formula transformations as simplification rules, along with two loop rules to handle loops. It turns out that rewriting by a rewrite system of this kind always terminates at a unique formula, independent of any order of rewriting. In the literature of rewrite systems (see, e.g., [5, 13]), this latter property is called the confluence property, and confluent and terminating systems are called canonical systems. A canonical system guarantees the termination at a unique expression independent of the order of rewriting. It is interesting to remark here that we could implement a form of nonmonotonic reasoning by a rewriting system, the two areas of research that had little connection previously. We show that these rewrite systems are sound and complete under the partial stable model semantics, in the sense that for any query, if it is written into $True$, then there must be a partial stable model containing this query, and if it is written into $False$, then there cannot be any partial stable model containing it. Since stable models are special cases of partial stable models, this means that if a query is written into $False$, there cannot be any stable model containing it. However, if it is written into $True$, in general, it could still happen that there may not be any stable model containing it. But our rewrite systems are of such nature that when a query is written into $True$, there will be a context, which is a set of literals, associated with the rewriting. To see whether there will be a stable model containing this query, one then only has to check whether this context can be extended to a stable model, a task that is normally much easier than finding a stable model from scratch. There is a special case, however, when the given propositional logic program is finite and so-called *odd-loop* free. For these programs, partial stable models coincide with stable models. Thus our rewrite systems are also sound and complete for these programs.

Then in Section 5, we extend the system to logic programs with abducibles, and again show that it is sound and complete under the partial stable model semantics. Again this implies that for any query, if a program is odd-loop free, the rewriting system generates a set of explanations that "covers" all possible explanations of the query. In the general case, the rewriting system generates an approximation of such a cover. Section 6 compares our rewriting system with other abductive procedures. In particular, we will see why SLDNF-like procedures cannot be adopted in a simple way for the kind of answer set programming advocated in this paper. The rewriting system presented in this paper has been implemented in Prolog. In Section 7 we discuss an application of our system to reasoning about actions and present some experimental results using our Prolog implementation.

Although our technical development is based on propositional programs, we will comment on how our rewriting framework can be used for classes of function-free programs for proving ground goals. One such class is the so-called domain restricted programs [27]. This material is given in Section 4.4.

2 Logic Programming Semantics

Here, we consider *(normal) logic programs* which are sets of rules of the form

$$a \leftarrow b_1, ..., b_m, \mathsf{not}\ c_1, ..., \mathsf{not}\ c_n$$

where $m, n \geq 0$, and a, b_i and c_i are atoms of a underlying propositional language L. Here an atom with "not " in front is called a *default negation*. As usual, a is called the *head* of the rule and the rest the *body* of the rule. For clarity of presentation, we may place a period at the end of a rule.

We sometimes write a rule of the above form as

$$a \leftarrow D, \mathsf{not}\_C$$

where D denotes the set of positive literals in the rule and not_C the set of default negations. Given such D and not_C, we use the notation, $\neg D = \{\neg\phi \mid \phi \in D\}$ and $C = \{\phi \mid \mathsf{not}\ \phi \in \mathsf{not}\_C\}$.

We sometimes also write a rule as $a \leftarrow B$, with B denoting the body of the rule. In this case, for convenience, we may also write B in a formula, as in the case of computing the Clark completion of a program, which will then stand for the conjunction of all the literals in B with the default negation not replaced by the negation operator "$\neg$" in our propositional language. When no confusion arises, the word $literal$ may refer to a classic literal as well as a default negation.

In this paper, we mainly deal with three semantics: the completion semantics [2], the stable model semantics [10], and the partial stable model semantics [29]. The stable model semantics has been generalized to the answer set semantics [11]. For the class of normal logic programs, which is the one considered in this paper, the two terminologies are interchangeable.

Given a propositional program P, the *Clark completion* of P, denoted $Comp(P)$, is the following set of equivalences: for each atom $\phi \in L$,

- if ϕ does not appear as the head of any rule in P, $\phi \leftrightarrow F \in Comp(P)$ (F stands for falsity here);
- otherwise, $\phi \leftrightarrow B_1 \vee ... \vee B_n \in Comp(P)$ (with default negations replaced by negative literals), if there are exactly n rules $\phi \leftarrow B_i \in P$ with ϕ as the head. We write T (tautology) for B_i if B_i is empty.

We now proceed to define partial stable models, which are originally defined in [29] under 3-valued logic. It is known that they can be defined equivalently in a number of different ways. Here, we adopt the definition given in [37] that does not rely on 3-valued logic (which will be introduced briefly at the end of this section).

We define some notations. Let E be a set of (classic) literals (or E may be a conjunction of literals in syntax). E^+ and E^- denote the subsets of positive literals and negative literals in E, respectively, and $E^{\mathbf{N}} = \{\mathsf{not}\,\phi \mid \neg\phi \in E\}$.

Let P be a program and ξ denote a proposition. For any set S of default negations, let

$$F_P(S) = \{\mathsf{not}\,\xi \mid P \cup S \not\vdash \xi\}$$

where $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\,\phi$ being treated as a named atom not_ϕ. It is easy to check that the function that applies F_P twice, denoted F_P^2, is monotonic: $S_1 \subseteq S_2$ implies $F_P^2(S_1) \subseteq F_P^2(S_2)$.

A set of literals M is a *partial stable model* of P iff it satisfies $F_P^2(M^{\mathbf{N}}) = M^{\mathbf{N}}$, $M^{\mathbf{N}} \subseteq F_P(M^{\mathbf{N}})$, and $M^+ = \{\xi \mid P \cup M^{\mathbf{N}} \vdash \xi\}$. Any atom ξ such that $\xi \notin M$ and $\neg\xi \notin M$ represents that ξ is undefined in M. The condition $M^{\mathbf{N}} \subseteq F_P(M^{\mathbf{N}})$ is to guarantee consistency. For example, with $P = \{p \leftarrow \mathsf{not}\,q.\ q \leftarrow \mathsf{not}\,p.\}$, the set of default negations $S = \{\mathsf{not}\,p, \mathsf{not}\,q\}$ satisfies $F_P^2(S) = S$, but not $S \subseteq F_P(S)$.

Let M be a partial stable model. M^+ is called an *answer set* (or a *stable model*) iff $F_P(M^{\mathbf{N}}) = M^{\mathbf{N}}$.

In general, a program may not have any answer sets. For instance, if $P = \{p \leftarrow \mathsf{not}\,p.\ q \leftarrow\}$, then P has no answer set, because there is no way to assign the values $true$ or $false$ to p. However, P has a partial stable model, $M = \{q\}$, in which q is true and p is undefined. This can be seen as follows (note that $M^{\mathbf{N}} = \emptyset$): $F_P(M^{\mathbf{N}}) = \{\mathsf{not}\,p\}$, $F_P(\{\mathsf{not}\,p\}) = M^{\mathbf{N}}$, $M^{\mathbf{N}} \subseteq F_P(M^{\mathbf{N}})$, and $M^+ = \{\xi \mid P \cup M^{\mathbf{N}} \vdash \xi\}$.

As our discussion sometimes refers to 3-valued logic, it is convenient to introduce it here briefly. This materail may be skipped if the reader is not going to read the proofs of the soundness and completeness theorems.

In 3-valued logic [17], there are three truth values: $true$, $false$, and $undefined$, denoted $\mathbf{t}$, $\mathbf{f}$, and $\mathbf{u}$ respectively. An interpretation I is a consistent set of literals: for any atom ϕ in the underlying language L, ϕ is true in I if $\phi \in I$, ϕ is false in I if $\neg\phi \in I$, and ϕ is undefined otherwise. The order of the truth values is defined as: $\mathbf{f} < \mathbf{u} < \mathbf{t}$. The connective $\neg$ (as well as not) is defined as: $\neg\mathbf{t} = \mathbf{f}$ ($\mathsf{not}\,\mathbf{t} = \mathbf{f}$), $\neg\mathbf{f} = \mathbf{t}$ ($\mathsf{not}\,\mathbf{f} = \mathbf{t}$), and $\neg\mathbf{u} = \mathbf{u}$ ($\mathsf{not}\,\mathbf{u} = \mathbf{u}$). The truth value of a conjunction is defined as the minimum value among the truth values of literals in the conjunction whereas the truth value of a disjunction is defined as the maximum value of the literals in the disjunction. The truth value of an implication $A \leftarrow B$ is $\mathbf{t}$ if the truth value of A is greater than or equal to that of B, otherwise it is $\mathbf{f}$. Logic equivalence is defined as: $A \leftrightarrow B$ iff $(A \rightarrow B) \wedge (B \rightarrow A)$.

We say that a 3-valued interpretation I satisfies a formula if the truth value of the formula is $\mathbf{t}$ in I. Thus a program rule $a \leftarrow B$ is satisfied by a 3-valued interpretation I if it is satisfied as an implication. A model of a program is an interpretation in which all the program rules are satisfied. We will use $I(Q)$ to denote the truth valuation of formula Q under 3-valued interpretation I. A 3-valued interpretation I reduces to a 2-valued interpretation if for every atom $\phi \in L$, either $\phi \in I$ or $\neg\phi \in I$. As usual, we use a set of atoms to denote a 2-valued interpretation.

3 Abduction in Logic Programming Revisited

In this section, we present a new definition of abduction in logic programming based on the answer set semantics. Let P be a logic program, A a set of propositions standing for abducibles, and q a proposition. In the following, without loss of generality, we shall assume that none of the abducibles in A occur in the head of a rule in P.[1]

In the following, by a *hypothesis* α we mean a consistent set of literals over A, i.e. it is not the case that p and $\neg p$ are both in α for some $p \in A$. We say that a hypothesis is *complete* if for each atom $p \in A$, either p or $\neg p$ is in α, but not both. Notice that a complete hypothesis is really a truth-value assignment over the language A. We say that a hypothesis α is an *extension* of another one β if $\beta \subseteq \alpha$, and a complete extension if it is an extension that is complete.

[1] If $p \in A$ occurs in the head of a rule, then we can always introduce a new proposition, say p', add the rule $p \leftarrow p'$ to P, add p' to A and delete p from A.

Definition 3.1 *A complete hypothesis α is said to be an explanation of q w.r.t. P and A iff there is an answer set M of $P \cup \alpha^+$ such that M contains q and for any $\neg p \in \alpha$, $p \notin M$, where α^+ is the set of atoms in α.*

Definition 3.2 *A hypothesis is said to be an explanation of q iff every complete extension of it is an explanation. A hypothesis α is said to be a minimal explanation if it is an explanation, and there is no other explanation α' such that $\alpha' \subset \alpha$.*

Consider the logic program P in Introduction about $canCross$. The following are the complete hypotheses that explain $canCross$:

$$\{boat, \neg leaking, \neg hasBucket\},$$
$$\{boat, \neg leaking, hasBucket\}, \{boat, leaking, hasBucket\}.$$

Now consider $\{boat, hasBucket\}$. Clearly every complete extension of this set is an explanation, so it is an explanation as well. Furthermore, it is a minimal explanation as none of its element can be deleted for it continue to be an explanation. Similarly, $\{boat, \neg leaking\}$ is also a minimal explanation.

If we take a hypothesis to be the conjunction of its elements, then we have that in propositional logic,

$$\bigvee_{\alpha \in \mathcal{S}_1} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_3} \alpha$$

where $\mathcal{S}_1$ is the set of all complete hypotheses that are explanations of q, $\mathcal{S}_2$ the set of all explanations of q, and $\mathcal{S}_3$ the set of all minimal explanations of q. Therefore the set of minimal explanations is a succinct representation of the set of all explanations.

It is clear from our definition that a complete hypothesis α is an explanation of q iff α^+ is an abductive explanation of q according to Kakas and Mancarella's definition. This implies that if none of the abducibles occur in the head of any clauses in P, then

$$\bigvee_{S \in \mathcal{S}_1} cl(S) \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha,$$

where $\mathcal{S}_1$ is the set of all abductive explanation of q according to Kakas and Mancarella's definition, $cl(S) = S \cup \{\neg p \mid p \in A, p \notin S\}$, and $\mathcal{S}_2$ is the set of all minimal explanations of q.

So in a sense, the set of Kakas and Mancarella's abductive explanations and that of our minimal explanations are equivalent. However, as we have seen above, the number of abductive explanations can be very large. Enumerating them all is impossible even in simple, small domains. In contrast, the number of minimal explanations is much smaller. More importantly, just like explanations in propositional logic, they only include "relevant propositions."

But computationally, it may be hard to compute minimal explanations from scratch. It is often easier to compute first a small "cover" of all explanations.

Definition 3.3 *A set $\mathcal{S}$ of hypotheses is said to be a* cover *of q w.r.t. P and A iff*

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_0} \alpha,$$

where $\mathcal{S}_0$ is the set of minimal explanations of q.

Proposition 3.4 *If $\mathcal{S}$ is a cover of q, then each $\alpha \in \mathcal{S}$ must be an explanation of q.*

Proof: If α' is a completion of α, then α' must be a completion of a minimal explanation of q, so must be an explanation of q. So it follows from the definition that α is an explanation. ∎

So a cover is a set of explanations such that any complete explanation must be an extension of one of the explanations in the cover. Once we have a cover, then we can find all minimal explanations by propositional reasoning alone.

To that end, we first prove the following result:

Proposition 3.5 *Let α be a hypothesis, and $\mathcal{S}$ a cover of q. We have that $\alpha \models \bigvee_{\beta \in \mathcal{S}} \beta$ iff α is an explanation of q.*

Proof: Only if case: suppose $\alpha \models \bigvee_{\beta \in \mathcal{S}} \beta$. This means that any complete extension of α must be an extension of one of the explanations in $\mathcal{S}$, so an explanation of q. It follows then that α is an explanation.
If case: if α is an explanation, and α' a complete extension of it. Then α' is an explanation. So it must be an extension of a minimal explanation. Because $\mathcal{S}$ is a cover, so α' must entail one of the explanations in $\mathcal{S}$. Thus $\alpha' \models \bigvee_{\beta \in \mathcal{S}} \beta$. Since α' is an arbitrary complete extension of α, so α must entail $\bigvee_{\beta \in \mathcal{S}} \beta$ as well. ∎

Recall that a conjunction of literals α is a *prime implicant* of a formula φ if $\alpha \models \varphi$, and there is no other β such that $\beta \models \varphi$ and β is a subset of α, i.e. α is a minimal conjunction of literals that entails φ.

From the last proposition, we immediately have the following:

Proposition 3.6 *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha \in \mathcal{S}} \alpha$.*

4 Goal Rewrite Systems

Having defined the notion of abductive explanations and covers of a query, we now consider the computational problem of how to actually generate them. To this end, we shall formulate a goal-oriented proof procedure based on confluent and terminating rewriting, and show that the procedure is sound and complete under the partial stable model semantics. To simplify our presentation, we shall consider first the case where there are no abducibles.

The idea of goal rewriting is simple. Given a program P, a completed definition $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ can be used as a rewrite rule from left to right: ϕ is rewritten to $B_1 \vee \ldots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \ldots \wedge \neg B_n$. We call these *literal rewriting*, and the completed definitions *program (rewrite) rules*.

In general, a *goal (formula)* is just a formula in our propositional language. However, in the following, all goals are assumed to be *signed*, which means that in these formulas, negation occurs only in front of atoms[2]. As we shall see, this restriction is necessary in the rewrite rules that we have for handling loops. It is easy to see that we do not lose any generality with this restriction as any goal can be equivalently transformed into a signed goal using the following rewrite rules: for any formulas Φ and Ψ,

$$\neg\neg\Phi \rightarrow \Phi$$
$$\neg(\Phi \vee \Psi) \rightarrow \neg\Phi \wedge \neg\Psi$$
$$\neg(\Phi \wedge \Psi) \rightarrow \neg\Phi \vee \neg\Psi$$

Note that, since we are dealing with partial stable models, 3-valued logic as introduced in Section 2 will serve as the underlying logic in goal rewrite systems. For example, it can be verified easily that the two sides of each rule above are logically equivalent under 3-valued logic.

In addition to program rules from $Comp(P)$ for literal rewriting, we also have the following two types of rewrite rules:

- simplification rules to transform and simplify goals, and
- loop rules for handling loops.

[2] The notion of signed goals was introduced in [19] for a similar purpose.

We introduce these two types of rules in the next two subsections, following which goal rewrite systems are defined and their properties investigated.

4.1 Simplification rules

The simplification subsystem is formulated with a mechanism of loop handling in mind, which requires keeping track of literal sequences $g_0, ..., g_n$ that have been chosen for rewriting, i.e. each g_i, $0 < i \leq n$, is in the goal formula resulted from rewriting g_{i-1}. These sequences are what we called *rewrite chains* below, and are checked for loops. In addition to rewrite chains, we also need to keep track of the *context* when a literal is written into T (tautology). This is necessary in order to maintain the consistency of a derivation: For a conjunction to be provable, not only each conjunct needs to be proved, the contexts under which these conjuncts are proved need to be consistent. These two notions are central in formulating our goal rewrite systems, and are defined as follows:

- *Rewrite Chain:* Suppose a literal l is written by its definition $\phi \leftrightarrow \Phi$ where $l = \phi$ or $l = \neg\phi$. Then, each literal l' in Φ is generated in order to prove l. This ancestor-descendant relation is denoted by $l \prec l'$. A sequence $l_1 \prec \ldots \prec l_n$ is then called a *rewrite chain*, abbreviated as $l_1 \prec^+ l_n$. Notice that it is essential here that Φ be in the form of a signed goal, and that when $\neg p$ is in Φ, we have that $l \prec \neg p$ but not $l \prec p$.

- *Context:* A rewrite chain $g = g_0 \prec g_1 \prec \ldots \prec g_n = T$ records a set of literals $C = \{g_0, ..., g_{n-1}\}$ for proving g. We will write $T(\{g_0, ..., g_{n-1}\})$ and call C a *context*.

  For simplicity, we assume that whenever $\neg F$ is generated, it is automatically replaced by $T(C)$, where C is the set of literals on the corresponding rewrite chain, and $\neg T$ is automatically replaced by F.

Note that for every literal in any derived goal, the rewrite chain leading to it from a literal in the given goal is uniquely determined. As an example, suppose we have the following equivalences: $\{a \leftrightarrow \neg b \wedge \neg c, b \leftrightarrow q \vee \neg p\}$. We then have a rewrite sequence

$$a \to \neg b \wedge \neg c \to \neg q \wedge p \wedge \neg c.$$

For the three literals in the last goal, we have the following rewrite chains from a:

$$a \prec \neg b \prec \neg q, \quad a \prec \neg b \prec p, \quad a \prec \neg c.$$

A rewrite chain should not be confused with a *rewrite sequence*. The former describes a dependency relationship between literals for book keeping purposes whereas the latter is a sequence of rewrite steps between goal formulas.

Simplification Rules:

Let Φ_i's be any goal formulas, C a context, and l a literal.

SR1. $F \vee \Phi \rightarrow \Phi$

SR1'. $\Phi \vee F \rightarrow \Phi$

SR2. $F \wedge \Phi \rightarrow F$

SR2'. $\Phi \wedge F \rightarrow F$

SR3. $T(C_1) \wedge T(C_2) \rightarrow T(C_1 \cup C_2)$ if $C_1 \cup C_2$ is consistent

SR4. $T(C_1) \wedge T(C_2) \rightarrow F$ if $C_1 \cup C_2$ is inconsistent

SR5. $T(C) \wedge l \rightarrow F$ if $\neg l \in C$

SR5'. $l \wedge T(C) \rightarrow F$ if $\neg l \in C$

SR6. $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$

SR6'. $(\Phi_1 \vee \Phi_2) \wedge \Phi_3 \rightarrow (\Phi_1 \wedge \Phi_3) \vee (\Phi_2 \wedge \Phi_3)$ □

The simplification system is a nondeterministic transformation system. The primed version of a rule is its symmetric case. Rules SR1, SR1', SR2, SR2', SR6, and SR6' are about the logical equivalence between the two sides of a rule (in 2-valued as well as 3-valued logic). SR3 merges two contexts if they are consistent, otherwise SR4 makes it a failure to prove. SR5 and SR5' prevent generating an inconsistent context before literal l is even proved.

It can be seen intuitively that the effect of SR5 and SR5' is similar to that of SR4. In a sequential implementation, if no inconsistent literals are ever allowed by SR5 or SR5' in individual steps, the condition for SR4 is never met, rendering SR4 redundant. On the other hand, SR4 alone is sufficient to safeguard the consistency of generated contexts, and it is not restricted to a sequential implementation. These rules represent different ways to guarantee consistency providing flexibility for implementation.

Note that, in general, the proof-theoretic meaning of a goal formula may not be the same as the logical meaning of the formula. For example, the goal formula $a \vee \neg a$ (a tautology in classic logic) could well lead to an F if neither a nor $\neg a$ can be proved.

For goal rewriting that does not involve loops, the system described so far is sufficient.

Example 4.1 *Let P be*

$$g \leftarrow \mathsf{not}\ a.$$
$$a \leftarrow \mathsf{not}\ b, \mathsf{not}\ c. \quad a \leftarrow b, \mathsf{not}\ d.$$
$$b \leftarrow q. \qquad b \leftarrow \mathsf{not}\ p.$$

Then $Comp(P)$ is:

$$g \leftrightarrow \neg a, \quad a \leftrightarrow (\neg b \wedge \neg c) \vee (b \wedge \neg d), \quad b \leftrightarrow q \vee \neg p,$$
$$q \leftrightarrow F, \quad p \leftrightarrow F, \quad d \leftrightarrow F, \quad c \leftrightarrow F.$$

The rewrite sequence below is generated by focusing on the beginning part of a goal.

$$\begin{aligned}
& g \rightarrow \neg a \\
& \rightarrow (b \vee c) \wedge (\neg b \vee d) \\
& \rightarrow (q \vee \neg p \vee c) \wedge (\neg b \vee d) \\
& \rightarrow (F \vee \neg p \vee c) \wedge (\neg b \vee d) \\
& \rightarrow (\neg p \vee c) \wedge (\neg b \vee d) \\
& \rightarrow (T(C) \vee c) \wedge (\neg b \vee d) \quad \textit{where } C = \{g, \neg a, b, \neg p\} \\
& \rightarrow (T(C) \wedge (\neg b \vee d)) \vee (c \wedge (\neg b \vee d)) \\
& \rightarrow (T(C) \wedge \neg b) \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \qquad \text{\% apply SR5} \\
& \rightarrow F \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \\
& \rightarrow (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \\
& \rightarrow (T(C) \wedge F) \vee (c \wedge (\neg b \vee d)) \\
& \rightarrow F \vee (c \wedge (\neg b \vee d)) \\
& \rightarrow c \wedge (\neg b \vee d) \\
& \rightarrow F \wedge (\neg b \vee d) \\
& \rightarrow F
\end{aligned}$$

4.2 Loop rules

After a literal l is rewritten, it is possible that at some later stage either l or $\neg l$ appears again in a goal on the same rewrite chain. Thus, a loop is a rewrite chain $\{l_1, ..., l_n\}$ where $l_1 = l_n$, $l_n = \neg l_1$, or $l_1 = \neg l_n$. A loop analysis involves classifying all the cases of loops, and for each one, determining the outcome of a rewrite according to the underlying semantics.

To understand the effect of loop rules, it is convenient to construct a *dependency graph* of a program: for each rule $a \leftarrow b_1, ..., b_m, \mathsf{not}\ c_1, ..., \mathsf{not}\ c_n$ in the program, there is a positive edge from a to each b_i, $1 \leq i \leq m$, and a negative edge from a to each c_j, $1 \leq j \leq n$.

For the problem at hand, there are only four cases of loops. When $l_1 = \neg l_n$ (or $l_n = \neg l_1$), the sign has changed from the beginning of the loop to the end. This loop yields a path from l_1 to l_1 (or from l_n to l_n) in the program's dependency graph that has an *odd* number of negative edges. So we shall call them odd loops. Odd loops must fail as one cannot prove a proposition by assuming its complement.

When $l_1 = l_n$, either every l_i, $1 < i < n$, has the same sign as those of l_1 and l_n, or not. When all l_i have the same sign, this sign is either positive or negative. They are identified as two different cases here since they must be treated differently according to the semantics. Otherwise, there is at least one l_i, $1 < i < n$, whose sign differs from those of l_1 and l_n. In this case, from the program's dependency graph, a loop is formed that has an *even* number of negative edges. In answer set programming, even loops are often used as a mechanism to generate alternative candidate answer sets.

Definition 4.2 *Let $S = l_1 \prec^+ l_n$ be a rewrite chain.*

- *If $\neg l_1 = l_n$ or $l_1 = \neg l_n$, then S is called an* odd loop.
- *If $l_1 = l_n$, then*
  - *S is called* a positive loop *if l_1 and l_n are both atoms and each literal on $l_1 \prec^+ l_n$ is also an atom;*
  - *S is called* a negative loop *if l_1 and l_n are both negative literals and each literal on $l_1 \prec^+ l_n$ is also negative;*
  - *Otherwise, S is called an* even loop.

In all the cases above, l_n is called a loop literal.

It turns out that we only need two rewrite rules to handle all four cases.

Loop Rules: Let $g_1 \prec^+ g_n$ be a rewrite chain.

LR1. $g_n \rightarrow F$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a positive loop or an odd loop.

LR2. $g_n \rightarrow T(\{g_1, ..., g_n\})$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a negative loop or an even loop. □

Apparently, a loop literal should always be rewritten by a loop rule. Our definition of a rewrite sequence below will ensure that for each loop literal, there is exactly one loop rule that can be applied to it.

Example 4.3 $P_1 = \{b \leftarrow \mathsf{not}\ c.\ \ c \leftarrow c.\}$. *Below, b is proved due to a negative loop, and the proof for $\neg b$ is failed due to a positive loop:*

$$b \to \neg c \to \neg c \to T(\{b, \neg c\})$$
$$\neg b \to c \to c \to F$$

$P_2 = \{d \leftarrow \mathsf{not}\ a.\ \ a \leftarrow \mathsf{not}\ b.\ \ b \leftarrow \mathsf{not}\ a.\}$. *Both d and $\neg d$ are proved, due to an even loop in either case:*

$$d \to \neg a \to b \to \neg a \to T(\{d, \neg a, b\})$$
$$\neg d \to a \to \neg b \to a \to T(\{\neg d, a, \neg b\})$$

$P_3 = \{a \leftarrow \mathsf{not}\ b.\ \ b \leftarrow \mathsf{not}\ b.\}$. *Neither a nor $\neg a$ is proved due to an odd loop:*

$$a \to \neg b \to b \to F$$
$$\neg a \to b \to \neg b \to F$$

4.3 Goal rewrite systems and their properties

In this subsection, we show that the goal rewrite systems defined above are confluent and terminating, and they are sound and complete under the partial stable model semantics.

Consider a propositional language L that consists of a (finite or infinite) number of propositions and their negative counterparts, all of which have been called *literals*, and special symbols F and $T(C)$ for each consistent set of literals C. Programs P in the language and their completions $Comp(P)$ are defined as usual, with the exception that any T appearing in $Comp(P)$ is replaced by $T(\emptyset)$. The set of *goals* $\mathcal{Q}_L$ consists of formulas constructed inductively from literals and special symbols by $\vee$ and $\wedge$. An *initial goal*, or *given goal*, is one without special symbols.

A *rewrite sequence* is a sequence of zero or more rewrite steps $Q_0 \to \ldots \to Q_k$, denoted $Q_0 \to^* Q_k$, such that Q_0 is an initial goal, and for each $0 \leq i < k$, Q_{i+1} is obtained from Q_i by

- literal rewriting at a non-loop literal in Q_i, or
- applying a simplification rule to Q_i or a subformula in Q_i, or
- applying a loop rule to a loop literal in Q_i.

Notice that since literal rewriting is done only on non-loop literals, once a loop is formed during a rewriting process, it has to be eliminated by a loop rule. So it is not possible to have a rewrite chain that contains more than one loop in any rewrite sequence.

Without loss of generality, we often assume that an initial goal is just a literal. In addition, we may call a subsequence $Q_i \to^* Q_k$ a rewrite sequence in the understanding that it is part of some rewrite sequence $Q_0 \to^* Q_i \to^* Q_k$ from an initial goal Q_0.

Definition 4.4 *A* goal rewrite system *is a triple* $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$*, where* $\mathcal{Q}_L$ *is the set of all goals,* $\mathcal{R}_P$ *is a set of rewrite rules which consists of program rules from* $Comp(P)$*, the simplification rules and the loop rules, and* $\to$ *is the set of all rewrite sequences.*

Goal rewrite systems are like term rewriting systems [5] everywhere except at terminating steps: a terminating step at a subgoal may depend on the history of rewriting.

A set of rewrite sequences defines a binary relation, say R, on the set of goal formulas: $R(Q, Q')$ iff $Q \to^* Q'$. Hence, a set of rewrite sequences corresponds to a binary relation.

Two desirable properties of rewrite systems are the properties of termination and confluence. Rewrite systems that possess both of these properties are called canonical systems. A canonical system guarantees that the final result of rewriting from any given goal is unique, independent of any order of rewriting.

Definition 4.5 *A goal rewrite system* $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ *is* terminating *iff there exists no endless rewrite sequence* $Q_1 \to Q_2 \to Q_3 \to$ *...... in* $\to$*.*

Definition 4.6 *Given a goal rewrite system, a goal is called a* normal form *if it cannot be rewritten by any rewrite rule.*

Since the simplification system is terminating and literal rewriting only generates non-repeated rewrite chains, it is clear that a goal rewrite system is terminating when the given program is finite. Because any literal has a program rule in $Comp(P)$, no literal will appear in any normal form. Furthermore, as the simplification system transforms a goal eventually to a disjunctive normal form (by SR6 and SR6'), goal rewriting always terminates at either F, or $T(C_1) \vee \ldots \vee T(C_m)$ for some $m \geq 1$. The latter indicates one or more ways by which the given goal is proved. We therefore have

Proposition 4.7 *Let* $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ *be a goal rewrite system. If* P *is finite then every rewrite sequence in* $\to$ *is finite. Further, for any rewrite sequence* $Q_0 \to^* Q_k$*, if* Q_k *is a normal form, then either* $Q_k = F$ *or* $Q_k = T(C_1) \vee \ldots \vee T(C_m)$*, for some* $m \geq 1$*.*

The confluence property is less obvious.

Definition 4.8 *A goal rewrite system* $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ *is* confluent *iff for any rewrite sequences* $t_1 \to^* t_2$ *and* $t_1 \to^* t_3$*, there exist* $t_4 \in \mathcal{Q}_L$ *and rewrite sequences* $t_2 \to^* t_4$ *and* $t_3 \to^* t_4$*.*

Theorem 4.9 *Any goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ with a finite P is confluent.*

Since the techniques in proving this theorem are of little relevance to the rest of this paper, we postpone the proof to Appendix.

The next theorem states the soundness and completeness of goal rewrite systems.

Theorem 4.10 *Let P be a finite program and $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ a goal rewrite system.*

Soundness: *For any literal g and any rewrite sequence $g \rightarrow^* T(C_1) \vee \ldots \vee T(C_m)$, there exists a partial stable model M_i of P, for each $i \in [1..m]$, such that $g \in C_i \subseteq M_i$.*

Completeness: *For any literal g true in a partial stable model M of P, there exists a rewrite sequence $g \rightarrow^* T(C_1) \vee \ldots \vee T(C_m)$ such that there exists $i \in [1..m]$, $g \in C_i \subseteq M$.*

Intuitively, the soundness says that whenever a literal g is proved, there exists a partial stable model containing g, and the completeness says that if g is true in one partial stable model, then there always exists a demonstrating proof. Notice that since stable models are a special case of partial stable models, this means that if g is true in some stable model, then there is a proof of it in the rewrite system. However, the converse is not true in general because stable model is a global notion and our rewrite system only checks local consistency. We shall have more to say about this after we generalize the theorem to include abducibles.

Before we prove the theorem, let's discuss the relationship between goal rewriting and derivability. Given a program P and a set of default negations Δ, $P \cup \Delta$ is viewed as a positive program. By a derivation of an atom ϕ using program P, we mean the usual least fixpoint construction of $P \cup \Delta$. In the following discussion, P refers to some fixed, finite program, and $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ is the goal rewrite system w.r.t. P. Δ and Δ' denote sets of default negations.

We consider rewriting without using loop rules, in which case loop literals are just left as terminating nodes. Any generated $T(C)$ is viewed as a conjunction of the literals in C. Goal rewriting in this case preserves logical equivalence under 2-valued as well as 3-valued logic – any generated equivalence logically follows from $Comp(P)$. For this reason, we are free to use $\leftrightarrow$ to express a rewrite sequence generated without using loop rules.

Suppose there are n rules in P with atom a as the head: $a \leftarrow B_i, \mathtt{not\_}C_i$, $i \in [1..n]$ (recall that B_i denotes the set of positive literals in the body and $\mathtt{not\_}C_i$ the set of default negations). The completion of a is then:

$$a \leftrightarrow (B_1 \wedge \neg C_1) \vee ... \vee (B_n \wedge \neg C_n)$$

The disjunctive normal form (DNF) at the right hand side of $\leftrightarrow$ describes all the possible ways to derive a using program P. For negative literal $\neg a$, we have

$$\neg a \leftrightarrow (\neg B_1 \vee C_1) \wedge ... \wedge (\neg B_n \vee C_n) \leftrightarrow \Psi_1 \vee ... \vee \Psi_m$$

where each Ψ_j is a conjunction of n literals $l_1 \wedge ... \wedge l_n$ where l_i is taken from $(\neg B_i \vee C_i)$. By the confluence property, such a DNF can always be obtained by repeatedly applying the distribution rules SR6 and SR6'. Intuitively, such a DNF expresses that any possibility of deriving a using P is *blocked* if we can demonstrate a derivation of each positive literal in Ψ_j, and show that it is impossible to derive l for any negative literal $\neg l$ in Ψ_j. In addition, we must ensure the consistency of Ψ_j. Without it, blocking in the sense above is ineffective, e.g., for the program $\{p \leftarrow q.\ \ p \leftarrow \texttt{not}\ q.\}$.

Since any literal l can be expressed equivalently by a DNF, $l \leftrightarrow \Psi_1 \vee ... \vee \Psi_m$, in the following, we say that l is defined by $\Psi_1 \vee ... \vee \Psi_m$ (or, $\Psi_1 \vee ... \vee \Psi_m$ is the definition of l).

Because of the termination and confluence properties, goal rewriting can be carried out in any order. Here, we consider a particular order of rewriting under which the semantical implications are easier to understand:

> Repeatedly, perform a literal rewriting and then transform the derived goal to a DNF.

Any $T(C)$ in such a DNF is viewed as a conjunction of the literals in C. As an illustration, suppose we have $g \leftrightarrow \Psi_1 \vee ... \vee \Psi_m$. Without loss of generality, suppose the next literal rewriting takes place at l_1 of $\Psi_1 = l_1 \wedge ... \wedge l_n$. Let the definition of l_1 be $\Phi_1 \vee ... \vee \Phi_s$. We then get the next DNF

$$\begin{aligned} & \Psi_1 \vee ... \vee \Psi_m \\ \leftrightarrow\ & [(\Phi_1 \vee ... \vee \Phi_s) \wedge \textstyle\bigwedge_{i\in[2..n]} l_i] \vee \Psi_2 \vee ... \vee \Psi_m \\ \leftrightarrow\ & (\Phi_1 \wedge \textstyle\bigwedge_{i\in[2..n]} l_i) \vee ... \vee (\Phi_s \wedge \textstyle\bigwedge_{i\in[2..n]} l_i) \vee \Psi_2 \vee ... \vee \Psi_m \end{aligned}$$

Goal rewriting in this fashion terminates at a DNF where every literal therein is a loop literal (after removing any disjunct that contains an F). To complete the process, we only need to apply loop rules, deal with conjunctive contexts according to simplification rules SR3 and SR4, and remove any conjunction that contains an F according to rules SR1-2 and SR1'-2'. Let us call such a DNF a *pre-normal form*. Clearly, a pre-normal form is unique.

Along with rewrite chains, a DNF represents a collection of *derivation trees*.

Definition 4.11 *Let g be a literal, and $g \leftrightarrow \Psi_1 \vee ... \vee \Psi_m$ be a rewrite sequence (without using loop rules) where each Ψ_i, $i \in [1..m]$, is a conjunction of literals and $T(C)$'s. The*

Figure 1: Derivation trees

derivation tree *(*d-tree *for abbreviation) of Ψ_i is a tree with g as the root node, and literals and $T(C)$'s in Ψ_i as the leaf nodes, where l' is a child node of l iff $l \prec l'$, or $l' = T(C)$ and $l \to T(C)$.*

In a derivation tree, each branch from g to a leaf node l corresponds to the rewrite chain of l. It's called a *derivation* tree because the relation between a positive node p and the collection of its child nodes Φ corresponds to a derivation of p, i.e., $P \cup \Phi^+ \cup \Phi^{\mathbf{N}} \vdash p$. This derivation relation is transitive over positive literals. For a negative literal $\neg p$, the collection of its child nodes Φ' expresses blocking of the derivation of p.

Note that a derivation tree of Ψ_i is unique, up to re-ordering of child nodes.

Example 4.12 *Consider the following program:*

$$\begin{array}{ll} g \leftarrow d, \textsf{not}\ c. & b \leftarrow b. \\ c \leftarrow \textsf{not}\ g, \textsf{not}\ e. & e \leftarrow e. \\ c \leftarrow b. & d. \end{array}$$

The pre-normal form from g is generated as:

$$g \leftrightarrow (T(\{g, d\}) \land g \land \neg b) \lor (T(\{g, d\}) \land e \land \neg b).$$

Figure 1 depicts the two derivation trees of g, one for each conjunction in the pre-normal form. For instance, the derivation tree at the left has two loop literals, one of which is on an even loop and the other on a negative loop. After applying the loop rules, the tree at the left produces $T(\{g, d, \neg c, \neg b\})$, whereas the one at the right generates an F, due to a positive loop.

In the following, let $g \leftrightarrow \Psi_1 \lor ... \lor \Psi_m \to^* T(D_1) \lor ... \lor T(D_s)$ where $\Psi_1 \lor ... \lor \Psi_m$ is a pre-normal form. Each Ψ_j, $j \in [1..m]$, is reduced either to an F, or to a $T(D_i)$ if each loop literal in Ψ_j is either on a negative loop or on an even loop, and the union of all conjunctive contexts is consistent. Clearly, since some Ψ_j may have reduced to an F, we have $s \leq m$. By the assumption of the above rewrite sequence, we know $s \geq 1$. Now suppose $\Psi_j \to^* T(D_i)$. We then have the following two lemmas.

Lemma 4.13 *For each* $i \in [1..s]$, $D_i^{\mathbf{N}} \subseteq F_P(D_i^{\mathbf{N}})$, $P \cup D_i^{\mathbf{N}} \vdash D_i^+$, *and* $P \cup F_P(D_i^{\mathbf{N}}) \vdash D_i^+$.

Proof: Assume $\mathsf{not}\, \xi \in D_i^{\mathbf{N}}$. Since D_i is consistent, $P \cup D_i^{\mathbf{N}} \not\vdash \xi$, hence $\mathsf{not}\, \xi \in F_P(D_i^{\mathbf{N}})$, and $D_i^{\mathbf{N}} \subseteq F_P(D_i^{\mathbf{N}})$. It is clear that for any positive literal p in D_i^+, $P \cup D_i^{\mathbf{N}} \vdash p$, as, in the derivation tree of Ψ_j, every rewrite chain (branch) from p either ends at a $T(C)$ where $C \subseteq D_i$, or is supported by a negative literal $\neg\xi$ for which $\mathsf{not}\, \xi \in D_i^{\mathbf{N}}$, i.e., $p \prec^+ \neg\xi$. It then follows from $D_i^{\mathbf{N}} \subseteq F_P(D_i^{\mathbf{N}})$ that $P \cup F_P(D_i^{\mathbf{N}}) \vdash p$, for any positive literal p. ■

Lemma 4.14 *For each* $i \in [1..s]$, $D_i^{\mathbf{N}} \subseteq F_P^2(D_i^{\mathbf{N}})$.

Proof: Suppose there are n rules in P with atom a as the head: $a \leftarrow B_i, \mathsf{not\_}C_i$, $i \in [1..n]$. For any $\mathsf{not}\, a \in D_i^{\mathbf{N}}$, to show $\mathsf{not}\, a \in F_P^2(D_i^{\mathbf{N}})$, we need to show $P \cup F_P(D_i^{\mathbf{N}}) \not\vdash a$.

For any Δ such that $P \cup \Delta \vdash a$, consider any minimal $\Delta' \subseteq \Delta$ such that $P \cup \Delta' \vdash a$. Δ' being minimal means that $P \cup (\Delta' - \{\mathsf{not}\, \xi\}) \not\vdash a$, for any $\mathsf{not}\, \xi \in \Delta'$. Thus, each of such Δ' corresponds to a particular derivation of a. Our goal is to show that for each of such Δ', there is at least one default negation $\mathsf{not}\, \xi \in \Delta'$ which is needed for the derivation of a but is not in $F_P(D_i^{\mathbf{N}})$.

It is clear that $\Delta' \neq \emptyset$, as otherwise $P \vdash a$, and thus there exists a rewrite chain, $\neg a \prec^+ F$, in the d-tree of Ψ_j, contradicting the assumption that $\Psi_j \to^* T(D_i)$.

Then, there is a derivation of a via some rule with a as the head relying on Δ', say $a \leftarrow B_k, \mathsf{not\_}C_k$, such that $\Delta' = \mathsf{not\_}C_k \cup \Delta''$ for some minimal Δ'' satisfying $P \cup \Delta'' \vdash B_k$. That Δ' is non-empty implies either $\mathsf{not\_}C_k$ is non-empty or Δ'' is non-empty. Let the definition of $\neg a$ be $\Pi_1 \vee ... \vee \Pi_s$, where each Π_j is of the form $l_1 \wedge ... \wedge l_n$, and for each $i \in [1..n]$, $l_i \in \neg B_i$ or $l_i \in C_i$. Consider the kth rule above with a as the head. If $l_k \in C_k$, l_k is positive and $\neg a \prec l_k$ is in the d-tree of Ψ_j. Otherwise, l_k is negative, say $l_k = \neg b$, and $l_k \in \neg B_k$. In this case, $\neg a \prec \neg b$ is in the d-tree of Ψ_j. By repeating the same argument for $\neg b$, and so on, since the d-tree of Ψ_j is finite, there are only two possibilities: every literal l such that $\neg b \prec^+ l$ is negative, or there exists a positive literal ξ such that $\neg b \prec^+ \xi$. Clearly, the former case implies $P \cup \Delta' \not\vdash b$. This is a contradiction to $P \cup \Delta' \vdash B_k$, where $b \in B_k$. Otherwise, there exists a positive literal ξ such that $\neg a \prec^+ \xi$. From Lemma 4.13, we know $P \cup D_i^{\mathbf{N}} \vdash \xi$, hence $\mathsf{not}\, \xi \notin F_P(D_i^{\mathbf{N}})$. Since for each minimal Δ' such that $P \cup \Delta' \vdash a$, there is at least one $\mathsf{not}\, \xi \in \Delta'$ that is needed for the particular derivation of a but is not in $F_P(D_i^{\mathbf{N}})$, we conclude $P \cup F_P(D_i^{\mathbf{N}}) \not\vdash a$. This completes the proof. ■

We are now ready to prove the soundness.

Proof of Soundness:

Suppose $g \rightarrow^* T(D_1) \vee \ldots \vee T(D_k)$. We show that, for each $i \in [1..k]$, D_i can be extended to a partial stable model. We know $D_i^{\mathbf{N}} \subseteq F_P(D_i^{\mathbf{N}})$ (Lemma 4.13), and $D_i^{\mathbf{N}} \subseteq F_P^2(D_i^{\mathbf{N}})$ (Lemma 4.14). We also know that F_P^2 is monotonic and F_P is anti-monotonic (i.e. $S_1 \subseteq S_2$ implies $F_P(S_1) \supseteq F_P(S_2)$). We therefore have the following two sequences

$$D_i^{\mathbf{N}} \subseteq F_P^2(D_i^{\mathbf{N}}) \subseteq F_P^4(D_i^{\mathbf{N}}) \ldots\ldots$$
$$F_P(D_i^{\mathbf{N}}) \supseteq F_P^3(D_i^{\mathbf{N}}) \supseteq F_P^5(D_i^{\mathbf{N}}) \ldots\ldots$$

such that $F_P^k(D_i^{\mathbf{N}}) \subseteq F_P^{k+1}(D_i^{\mathbf{N}})$, for any even number $k \geq 0$, and $F_P^j(D_i^{\mathbf{N}}) \subseteq F_P^{j+1}(D_i^{\mathbf{N}})$, for any odd number $j \geq 1$. Because P is finite, we can restrict the function F_P to the finite domain consisting of atoms appearing in P (any other atom in the underlying language is false in any partial stable model). Thus, the two sequences above converge at $F_P^n(D_i^{\mathbf{N}})$, for some even number $n \geq 0$, such that $F_P^n(D_i^{\mathbf{N}}) = F_P^{n+2}(D_i^{\mathbf{N}})$, $F_P^{n+1}(D_i^{\mathbf{N}}) = F_P^{n+3}(D_i^{\mathbf{N}})$, and $F_P^n(D_i^{\mathbf{N}}) \subseteq F_P^{n+1}(D_i^{\mathbf{N}})$. By definition, the 3-valued interpretation M that corresponds to the fixpoint $F_P^n(D_i^{\mathbf{N}})$, namely $M^- = \{\neg\phi \mid \mathsf{not}\,\phi \in F_P^n(D_i^{\mathbf{N}})\}$ and $M^+ = \{\phi \mid P \cup F_P^n(D_i^{\mathbf{N}}) \vdash \phi\}$, is a partial stable model of P. ■

We need the following lemma for the proof of completeness.

Lemma 4.15 *A partial stable model M of a program P is a 3-valued model of $Comp(P)$.*

Proof: For any atom a defined by $a \leftarrow Body_i$, $i \in [1..m]$, we show M satisfies its completion: $a \leftrightarrow Body_1 \vee \ldots \vee Body_m$. This can be proved by considering all three cases: $M(a) = \mathbf{t}$, $M(a) = \mathbf{f}$, and $M(a) = \mathbf{u}$. The first two cases are straightforward. For the last case, since M is a model of P, we know $M(Body_i) \neq \mathbf{t}$, for any i. We show $M(Body_i) = \mathbf{u}$, for some i. Now assume $M(Body_i) = \mathbf{f}$ for all i. We show that this leads to a contradiction. Under this assumption, there is $\phi \in Body_i$ such that $M(\phi) = \mathbf{f}$, for each i. ϕ is either an atom or a default negation. Suppose ϕ is an atom. Then that $M(\phi) = \mathbf{f}$ implies $\mathsf{not}\,\phi \in M^{\mathbf{N}}$. Since M is a partial stable model of P, we have $M^{\mathbf{N}} = F_P^2(M^{\mathbf{N}})$ hence $\mathsf{not}\,\phi \in F_P^2(M^{\mathbf{N}})$. By the definition of F_P, we have $P \cup F_P(M^{\mathbf{N}}) \not\vdash \phi$. If ϕ is a default negation $\mathsf{not}\,q$, then $M(q) = \mathbf{t}$ hence $P \cup M^{\mathbf{N}} \vdash q$. As a partial stable model, M satisfies $M^{\mathbf{N}} \subseteq F_P(M^{\mathbf{N}})$. Thus, $P \cup F_P(M^{\mathbf{N}}) \vdash q$. In either case, we have $P \cup F_P(M^{\mathbf{N}}) \not\vdash a$ hence $\mathsf{not}\,a \in F_P^2(M^{\mathbf{N}})$, i.e., $\mathsf{not}\,a \in M^{\mathbf{N}}$ and $\neg a \in M$. This contradicts the assumption that $M(a) = \mathbf{u}$. ■

Proof of Completeness:

Assume $g \in M$ for some partial stable model M. By Lemma 4.15, M is a 3-valued model of $Comp(P)$. Due to the confluence and termination properties, any literal can be reduced to a pre-normal form such that the equivalence $g \leftrightarrow \Psi_1 \vee ... \vee \Psi_m$ logically follows from $Comp(P)$. Below, $RC(\Psi_i)$ denotes the set of literals appearing on the rewrite chains of the literals in Ψ_i (and is viewed as a conjunction when appropriate). It follows from the transitivity of $\leftrightarrow$ that the following are equivalent:

- M satisfies g;
- M satisfies $RC(\Psi_1) \vee ... \vee RC(\Psi_m)$;
- M satisfies some consistent $RC(\Psi_i)$ (any inconsistent $RC(\Psi_j)$ can be removed);
- M satisfies some $RC(\Psi_i)$ for which Ψ_i is odd-loop free (any Ψ_j on an odd loop implies $RC(\Psi_j)$ is inconsistent).

Now suppose $\Psi_1 \vee ... \vee \Psi_m \rightarrow^* T(D_1) \vee ... \vee T(D_{m'})$. Then, M satisfies $D_1 \vee ... \vee D_{m'} \vee U$ where U is the disjunction of those Ψ_j that are consistent (i.e., each $RC(\Psi_j)$ is consistent) but on a positive loop. We are done if M satisfies $D_1 \vee ... \vee D_{m'}$. Assume M does not satisfy $D_1 \vee ... \vee D_{m'}$. Then M only satisfies U. Consider any Φ in U that is satisfied by M. For any loop literal ξ in Φ, since ξ is on a positive loop, we know $P \cup RC(\Phi)^{\mathbf{N}} \not\vdash \xi$. As M satisfies Φ, we have $RC(\Phi) \subseteq M$, hence $\xi \in M$ and $P \cup M^{\mathbf{N}} \vdash \xi$. It follows there exists a Δ such that $\Delta \cup RC(\Phi)^{\mathbf{N}} \subseteq M^{\mathbf{N}}$ and $P \cup \Delta \vdash \xi$. That is, ξ can be reduced, consistently with M, using an alternative rule with ξ as the head. From the definition of ξ, we know that no possibility of deriving ξ is missed. As each loop literal in Φ can be extended this way, we can replace the derivations from (the first occurrences of) such loop literals by these alternatives. Let Φ' be the conjunction whose d-tree is that of Φ except the derivations from such literals are removed. Then, there exists a rewrite sequence $\Phi' \rightarrow^* \Psi'$ such that Ψ' is in the pre-normal form and not on a positive loop, and $RC(\Psi')$ is consistent due to $RC(\Psi') \subseteq M$. Thus, $\Psi' \rightarrow^* T(C)$ where $C = RC(\Psi')$. Therefore, $T(C)$ must be one of the $T(D_i)$. This contradicts the assumption that M does not satisfy $D_1 \vee ... \vee D_{m'}$. We are done. ■

A partial stable model M is *maximal* if there is no other partial stable model M' such that $M \subset M'$. Maximal partial stable models minimize the undefined. A stable model is clearly a maximal partial stable model that has no undefined. Maximal partial stable models are also known as *regular models* and *preferred extensions* [6, 31, 37]. Clearly, any literal g is true in a partial stable model if and only if g is true in a maximal partial stable model. We therefore have

Corollary 4.16 *Goal rewrite systems are sound and complete w.r.t. the regular model semantics.*

4.4 Rewriting with non-ground programs

The rewriting framework introduced here is defined for ground programs. It however can be applied to function-free programs for proving ground goals. In abduction, observations are usually formulated as ground goals. The idea is that if every derived goal is ground, then all the mechanisms given in this paper become applicable by adding an unification algorithm (in fact, a simple instantiation process is sufficient). Obviously, if for every rule in the given program a variable that appears in the body also appears in the head, then a ground goal will be rewritten to another ground goal.

One class of programs that can easily satisfy this property is the so-called *domain restricted programs* [27]. The idea of domain restriction is that if every variable that appears in a rule appears in a positive body literal of the rule, and draws its value from a finite domain, then the instantiation of the rule can be restricted to these domain values. The interest in [27] is to instantiate a function-free program to a possibly smaller ground program while preserving the stable model semantics. For our purpose of non-ground rewriting over ground goals, a rule can be instantiated only on domain predicates for variables that do not appear in the head. For example, to describe the reachability from a node s in a graph we may write

$$reached(U) \leftarrow arc(s, U).$$
$$reached(U) \leftarrow arc(V, U), reached(V).$$

along with some facts about the predicate $arc(X, Y)$, which can be considered a domain predicate. Since in the second rule the variable V appears only in the body, we instantiate the rule to

$$reached(U) \leftarrow arc(a_i, U), reached(a_i).$$

for each node a_i such that $arc(a_i, U)$ is true for some U. Suppose there are n such nodes $a_1, ..., a_n$. Then, the completion of the predicate $reached$ is:

$$reached(X) \leftrightarrow$$
$$arc(s, X) \vee [arc(a_1, X) \wedge reached(a_1)] \vee ... \vee [arc(a_n, X) \wedge reached(a_n)].$$

Thus, a ground goal, say $reached(t)$ (to prove that t can be reached from s), is always rewritten to another ground goal.

5 Rewrite Systems for Abduction

The rewriting framework that we defined in the preceding section can be extended for abduction in a straightforward way: the only difference in the extended framework is that we do not apply the Clark completion to abducibles. That is, once an abducible appears in a goal, it will remain there unless it is eliminated by the simplification rule $SR2$ or $SR2'$.

As an abducible may appear in a goal positively or negatively, we need a terminology to refer to both of them: an *abducible literal* is either an abducible ϕ or its negative counterpart $\neg\phi$. Just like a rewrite to T is written as $T(C)$, where C is the underlying rewrite chain (cf. Section 4.1), a rewrite to an abducible literal l will be written as $l(C)$ where C is the rewrite chain leading to, and including l. Thus when we write $l(C)$, it is understood that C always contains l.

In the following we shall denote by $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the rewrite system obtained by the logic program P and the set A of abducibles. These rewrite systems are both sound and complete with respect to the partial stable models semantics.

Theorem 5.1 *Let P be a finite program, A a set of abducibles, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the goal rewrite system with respect to P and A.*

Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^* G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G',$$

where each l_i is either an abducible literal or T, if $C_1 \cup \cdots \cup C_k$ is consistent, then there exists a partial stable model M of $P \cup \{l_1, ..., l_k\}^+$ such that $g \in \bigcup_{1 \leq j \leq k} C_i \subseteq M$.

Completeness: *For any set of atoms $S \subseteq A$, and any literal g in a partial stable model M of $P \cup S$, there exists a rewrite sequence*

$$g \rightarrow^* G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G',$$

such that $g \in \bigcup_{1 \leq j \leq k} C_j \subseteq M$.

Proof: Since none of the abducibles appear in the head of any program rule, the statements here about soundness and completeness follow directly from those of Theorem 4.10. ■

We have again stated our results in terms of the partial stable model semantics. Again the reason that our rewriting system may not be sound under the stable model semantics is that stable models check for global consistency, but our system checks only local ones. There are several ways to make the rewriting system also sound and complete for stable models. When a conjunction of abducibles

$$l_1(C_1) \wedge \cdots \wedge l_k(C_k)$$

is generated, one can check if $C = C_1 \cup \cdots \cup C_k$ is consistent and complete. If it is, then $\{l_1, ..., l_k\}$ is an explanation. If it is consistent but not complete, then we can either call a stable model generator to see if C can be extended to a stable model or we can choose an atom p such that neither it nor its negation is in C, and continue the rewriting with either $p(C)$ or $\neg p(C)$, until a complete context is obtained.

There is however an important special class of logic programs where partial stable models and stable models coincide. We say that a program has no odd loops (odd-loop free) if there is no odd loop starting with any literal. Since goal rewrite systems are confluent, any odd-loop in the program's dependency graph can replicate itself in a rewrite chain of some goal rewrite sequence. Therefore, there is no essential difference between our notion of odd-loop free and the notion of *negative cycle free* in the literature [8, 32].

It has been shown in [36] that for any nonground, negative cycle free program with a well-founded stratification, its partial stable models are all 2-valued and thus coincide with its stable models.[3] A stratification in this case is a partial order of strata each of which contains ground atoms that are involved in some loops among themselves. In a well-founded stratification, there is no infinite descending chain in the partial order. That is, every such chain must have a base stratum.[4] This property allows us to construct, along the well-founded stratification in the bottom-up fashion, a 2-valued justifiable model (which is known to be a stable model) from any 3-valued justifiable model. In this way one can show there is no partial stable model with undefined atoms. Since finite propositional programs all have a well-founded stratification, for these programs our rewriting system becomes sound and complete under the stable model semantics. We thus have the following results.

Theorem 5.2 *Let P be a finite program, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ a goal rewrite system. Suppose q is a proposition and*

$$q \rightarrow^* [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee$$
$$[l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]$$

[3] The result was stated for maximal partial stable models. However, the claim can be extended to all partial stable models by exactly the same proof.

[4] Here is a program that has no well-founded stratification: $\{p(a).\ \ p(x) \leftarrow \mathsf{not}\, p(f(x)).\}$.

is a rewrite sequence such that each l_{ij} *is either* T *or an abducible literal, and* $C_{i1} \cup \cdots \cup C_{ik_i}$ *is consistent for each* i*. If* P *has no odd loops then*

$$\{\{l_{11}, \cdots, l_{1k_1}\}, \cdots, \{l_{m1}, \cdots, l_{mk_m}\}\}$$

is a cover of q*. In general, for arbitrary* P *we have*

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \supset [l_{11} \wedge \cdots \wedge l_{1k_1}] \vee \cdots \vee [l_{m1} \wedge \cdots \wedge l_{mk_m}]$$

where $\mathcal{S}$ *is any cover of* q*.*

Proof: We show that for each i, $\{l_{i1}, \cdots, l_{ik_i}\}$ is an explanation of q. Let's denote the set by α, and let β be any complete hypothesis that extends α. We need to show that there is an answer set of $P \cup \beta^+$ that includes q. (Notice again that we have assumed that none of the propositions in A appear in the head of any rule in P.) Since there is a rewrite sequence of the form:

$$q \to^* G \vee [l_{i1}(C_{i1}) \wedge \cdots \wedge l_{ik_i}(C_{ik_i})] \vee G'$$

in $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \to \rangle$, there is a rewrite sequence of the form $q \to^* T(C) \vee G$ in $\langle \mathcal{Q}_L, \mathcal{R}_{P\cup\beta^+}, \to \rangle$, where

$$C = C_{i1} \cup \cdots \cup C_{ik_i},$$

because $C_{i1} \cup \cdots \cup C_{ik_i}$ is consistent and $\{l_{i1}, ..., l_{ik_i}\}$ is a subset of β. Thus by Theorem 4.10 there is a partial stable model M of $P \cup \beta^+$ such that $q \in M$. Now since P does not have any odd loops, neither does $P \cup \beta^+$. So M^+ is an answer set of $P \cup \beta^+$. This shows that β is an explanation of q. So α is an explanation.

Now let β be any complete explanation of q. We need to show that for some i, $\{l_{i1}, \cdots, l_{ik_i}\}$ is a subset of β. Let M be an answer set of $P \cup \beta^+$ that includes q. By Theorem 4.10, there is a rewrite sequence of the form $q \to^* T(C) \vee G$ in $\langle \mathcal{Q}_L, \mathcal{R}_{P\cup\beta^+}, \to \rangle$. Then there must be a rewrite sequence of the form: $q \to^* [l_1(C_1) \wedge \cdots \wedge l_j(C_j)] \vee G'$ in $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \to \rangle$ such that $\{l_1, ..., l_j\} \subseteq T \cap \beta$, and $C_1, ..., C_j$ are consistent. So by Theorem 4.9, it must be the case that

$$l_1 \wedge \cdots \wedge l_j \models [l_{11} \wedge \cdots \wedge l_{1k_1}] \vee \ldots \vee [l_{m1} \wedge \cdots \wedge l_{mk_m}]$$

in propositional logic. But $\{l_1, \cdots, l_j\}$ is contained in β, which is a complete hypothesis, so there must be an i such that $\beta \models l_{i1} \wedge \cdots \wedge l_{ik_i}$. ■

Consider again the boat example in Section 1. The Clark completion of $canCross$ is:

$$canCross \equiv (boat \wedge \neg leak) \vee (boat \wedge leak \wedge hasBucket).$$

Since $boat$, $leak$ and $hasBucket$ are abducibles, rewriting for $canCross$ terminates in one step, and produces the following cover:

$$\{boat \wedge \neg leak,\ boat \wedge leak \wedge hasBucket\}.$$

Notice that the second explanation is not minimal. To get minimal ones, we have to compute prime implicants of the disjunction of explanations in the cover, which are $boat \wedge \neg leak$ and $boat \wedge hasBucket$.

6 Related Work

Traditionally, logic programming proof procedures have been defined abstractly in terms of derivation and refutation. Termination has been considered a separate, implementation issue. On the one hand, this separation is possible since the semantics that these procedures compute allow the completeness to be stated without resorting to termination. But completeness is rarely guaranteed in an implementation. On the other hand, the separation is also necessary since these procedures deal with non-ground programs for which the problem of loop-checking is undecidable (even for function-free programs [1]). However, for answer set programming where each answer set is taken as a solution to a given problem, loop handling becomes a semantic issue – a sound and complete backward chaining procedure cannot be defined without it.

A number of abductive procedures have been proposed for the two-valued as well as three-valued completion semantics [3, 4, 9], of which the system by Console et al. [3] and the IFF procedure by Fung and Kowalski [9] also use rewriting as the main mechanism to compute explanations. Console et al. show that, for non-recursive programs (called hierarchical programs), abductive explanations can be computed as a deduction by rewriting using iff-definitions. Fung and Kowalski extend this idea to the class of all normal programs, and get completeness results that can be stated without resorting to termination. This improves the completeness theorems by Denecker and De Schreye [4] which rely on termination as a condition. All of these procedures are defined for *cautious reasoning* – computing bindings for which an (existential) goal is true in all indented models. In our case the reasoning mode is *brave* – establishing whether a query is true in one of the intended models. For example, with the program $\{a \leftarrow \mathsf{not}\, b.\ b \leftarrow \mathsf{not}\ a.\}$, the answer to query a should be *no* in their case (however, none of these procedures actually terminates and returns this answer), and *true* in a stable model in our case. Apparently, the differences between the proof methods for consequence finding and those for brave reasoning lie in the correct handling of loops in the latter in order to capture each of the intended models.

Our work is closely related to another abductive procedure, the Eshghi-Kowalski procedure (EKP) [7] (also see [6]), which is sound and complete for ground programs under the finite-failure three-valued stable model semantics in which loops causing infinite failure are modeled by the truth value *undefined* [12]. It is known that with an appropriate handling of positive loops (distinguished as positive and negative loops in this paper), EKP can be made complete for the partial stable model semantics. To some extent, one can say that our goal rewriting system (GRS) simulates EKP in a nontrivial way.

1. GRS incurs no backtracking! Backtracking is simulated by rewriting disjunctions, e.g., $F \vee \Phi \rightarrow \Phi$.

2. Loops that go through negation are handled in EKP by nested structures while in GRS by a *flat* structure using rewrite chains.

These features plus loop handling made it possible to formalize our system as a rewriting system benefiting from the known properties of rewrite systems in the literature. (This further distinguishes our use of rewrite systems from the literature, e.g., in [9].) To illustrate these features, consider the following program

$$\begin{array}{ll} r1.\ g \leftarrow \mathsf{not}\, a. & r3.\ b \leftarrow a. \\ r2.\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c. & r4.\ c \leftarrow a. \end{array}$$

and the question whether we can prove g. We may answer this question by the following reasoning: To have g we must have $\mathsf{not}\ a$ (r1); to have $\mathsf{not}\ a$ we must have either b or c (r2) which requires having a (r3 and r4). This results in a contradiction. Therefore, g cannot be proved. Note that in this reasoning we need to remember what was required previously ($\mathsf{not}\ a$ in this case). This is exactly how the proof is done by GRS:

$$g \rightarrow \neg a \rightarrow b \vee c \rightarrow a \vee c \rightarrow F \vee c \rightarrow c \rightarrow a \rightarrow F$$

However, EKP will go through *six* nested levels, and do it twice through backtracking, before the same conclusion can be reached. That a single derivation branch in EKP could be deeply nested brings no surprise that any attempt to lay out a proof presents some challenge, even for small programs. We note that the mechanism of rewrite chain is indispensable in any implementation of a top-down procedure if termination and completeness are to be preserved. For GRS, such a mechanism is used both for termination and for the implementation of the semantics, resulting in a much simpler yet more natural proof structure.

Our goal rewriting procedure departures from the traditional SLDNF-like procedures also in the use of a *computational rule*. Recall that a computational rule is a function that returns a subgoal form a goal. Its interest originates from the so called *independence* of computational rules for Horn clause programs, which says that the commitment to any subgoal can be made without the need of looking back, because no solutions will be missed. For normal programs, such a computational rule must be *fair* which requires generation of an SLD-tree that is either finite, or every subgoal in it is *eventually* selected [25]. These conditions require a fair computation rule to be implemented by a form of breadth-first search in order to find a finitely failed tree. In contrast, since goal rewrite systems are confluent and terminating, literal selection can be arbitrary and is guaranteed to be fair. As an example, consider the following program:

$$\begin{array}{ll} g \leftarrow \mathsf{not}\ a. & \\ a \leftarrow \mathsf{not}\ b, \mathsf{not}\ c. & a \leftarrow \mathsf{not}\ d. \\ b \leftarrow \mathsf{not}\ d. & d \leftarrow \mathsf{not}\ e. \\ e \leftarrow \mathsf{not}\ d. & c. \end{array}$$

The program has two stable models, $\{c, d, g\}$ and $\{a, b, c, e\}$. A complete procedure for brave reasoning should generate a proof for g. Such a proof is reflected naturally, and logically, in goal rewriting

$$g \rightarrow \neg a \rightarrow (b \vee c) \wedge d \rightarrow (\neg d \vee c) \wedge d \rightarrow \ldots$$

Any of the literals in the last goal above can be selected for literal rewriting, or the goal can be transformed to $(\neg d \wedge d) \vee (c \wedge d)$. Even if we can prove $\neg d$, its conflict with d will fail this alternative.[5] Thus $(\neg d \wedge d) \vee (c \wedge d)$ will be rewritten to $F \vee (c \wedge d)$ and then to $(c \wedge d)$. Continuing, we have

$$\begin{array}{l} \rightarrow c \wedge d \\ \rightarrow T(\{g, \neg a, c\}) \wedge d \\ \rightarrow T(\{g, \neg a, c\}) \wedge \neg e \\ \rightarrow T(\{g, \neg a, c\}) \wedge d \\ \rightarrow T(\{g, \neg a, c\}) \wedge T(\{g, \neg a, d, \neg e\}) \\ \rightarrow T(\{g, \neg a, c, d, \neg e\}) \end{array}$$

In an SLDNF-procedure, for instance, in the Eshghi-Kowalski procedure, to prove g we need to prove that any attempt to prove a fails. The two possibilities of proving a are kept in a goal set

$$\{\leftarrow \mathsf{not}\ b, \mathsf{not}\ c; \quad \leftarrow \mathsf{not}\ d\}$$

[5]As a technical note, this example also explains why in general we cannot have a rewrite rule of the form $T(C) \vee \Phi \rightarrow T(C)$, even if we content with one proof.

Both should fail in order for g to succeed. From the first goal, suppose we choose $\texttt{not}\, b$. To fail this goal, we can get a derivation of b using $\texttt{not}\, d$; so far we have succeeded in choosing $\texttt{not}\, b$ for the current goal. However, this proof causes a conflict in order to fail the second goal by proving d. Thus, the choice of $\texttt{not}\, b$ in the first goal does not give us a proof that both goals fail. In fact, we must choose $\texttt{not}\, c$ in the first goal in order to fail both. Since in general we do not know which subgoal leads to a proof, if a procedure is non-terminating, a fair computational rule must explore all alternatives in an interleaving fashion in order not to miss an answer to a query.

7 Applications and Experimental Results

We have implemented the writing framework in SWI-Prolog. Our implementation adopts a strategy based on eager literal rewriting and lazy expansion, which resembles the familiar depth-first strategy. The main idea is to delay applying distribution rules SR6 and SR6' as much as possible to avoid an exponential blow up in goal size. We thus fix the order of rewriting by focusing on the leftmost literal of a goal. We say that a literal l in a goal Q is *rewritable* (for literal rewriting) if it is either at the leftmost position of Q, or at the second leftmost position with a conjunct $T(C)$ at the left where l is not a loop literal and neither l nor $\neg l$ is in C. That is, a literal l is rewritable only in goals that begin with one of the three forms: $l \vee \Phi$, $l \wedge \Phi$, and $T(C) \wedge l$, where Φ is a formula. If l is a loop literal then a loop rule is applied; if $\neg l \in C$ then rule SR5 is applied to produce an F; if $l \in C$, l is already proved, thus the rewrite chain of l is merged with C. Being lazy means that a goal is simplified only when doing so is necessary to make the goal rewritable. In particular, the distribution rules SR6 and SR6' will be applied only when a goal is not rewritable, and none of the above applies.

Under this strategy, we keep rewriting the literal at the leftmost position of a goal. Eventually, it will become an F or a $T(C)$. If it is a $T(C)$, we will keep rewriting the leftmost literal l in conjunction with $T(C)$, during which $\neg l \in C$ is checked for consistency. When this l is rewritten to $T(C')$, C and C' are merged; and recursively, we continue to focus on the literal at the leftmost position of the goal, or the one in conjunction with $T(C \cup C')$. The reader may refer to the rewrite sequence in Example 4.1 as an example, which is generated using this strategy.

In our current implementation, the input is required to be a set of Clark completion sentences, one for each non-abducible proposition. In the following, we discuss the performance of our implementation on one particular application of abduction in logic programming, which is the problem of computing successor state axioms from a causal action theory [22, 24, 23].

Consider a logistics domain in which we have a truck and a package. We know that the truck and the package can each be at only one location at any given time, and that if the package is in the truck, then when the truck moves to a new location, so is the package. Suppose that we have the following propositions:

- ta(x) – the truck is at location x initially;
- pa(x) – the package is at location x initially;
- in – the package is in the truck initially;
- ta(x, y, z) – the truck is at location x after the action of moving it from y to z is performed;
- pa(x, y, z) – the package is at location x after the action of moving the truck from y to z is performed; and
- in(y, z) – the package is in the truck after the action of moving the truck from y to z is performed.

We then have the following logic program:

$$\begin{array}{l}
\text{ta}(X, X1, X). \\
\text{pa}(X, X1, X2) \leftarrow \text{ta}(X, X1, X2), \text{in}(X1, X2). \\
\text{ta}(X, X1, X2) \leftarrow X \neq X2, \text{ta}(X), \mathsf{not}\ \text{taol}(X, X1, X2). \\
\text{taol}(X, X1, X2) \leftarrow Y \neq X, \text{ta}(Y, X1, X2). \\
\text{pa}(X, X1, X2) \leftarrow \text{pa}(X), \mathsf{not}\ \text{paol}(X, X1, X2). \\
\text{paol}(X, X1, X2) \leftarrow Y \neq X, \text{pa}(Y, X1, X2). \\
\text{in}(X, Y) \leftarrow \text{in}.
\end{array}$$

The first rule is the effect axiom. The second rule is a causal rule which says that if a package is in the truck, then the package should be where the truck is. The rest are frame axioms. For instance, the third one is the frame axiom about ta, with the help of a new predicate taol: if the truck is initially at X, and if one cannot prove that it will be elsewhere after the action is performed, then it should still be at X.

As one can see, the above program, when fully instantiated over any given finite set D of locations, has no odd loops. So our rewrite system will generate a cover for any query. Note that in the program we have omitted domain predicate $loc(X)$ for each variable X in the body of a rule (all the variables in the program refer to locations). Thus, the program is domain restricted, and we only need to instantiate the variable Y in the fourth and sixth rules over the domain of locations.

Now let the set A of abducibles be the following set:

$$\{\text{in}\} \cup \{\text{pa}(x), \text{ta}(x) \mid x \in D\}.$$

The following table shows some of the results for $D = \{1, 2, 3, 4\}$:[6]

Query	Result	Time
ta$(1, 2, 3)$	false	0.0
ta$(3, 2, 3)$	true	0.0
pa$(1, 2, 3)$	pa$(1) \wedge \neg$in	0.05
$\neg$pa$(1, 2, 3)$	$\neg$pa$(1) \vee$ in $\vee$ pa$(2) \vee$ pa$(3) \vee$ pa(4)	0.2
pa$(2, 2, 3)$	pa$(2) \wedge \neg$in	0.08
$\neg$pa$(2, 2, 3)$	$\neg$pa$(2) \vee$ in $\vee$ pa$(1) \vee$ pa$(3) \vee$ pa(4)	0.1
pa$(3, 2, 3)$	pa$(3) \vee$ in	0.25
$\neg$pa$(3, 2, 3)$	$\neg$in $\wedge \neg$pa$(3) \vee \neg$in $\wedge$ pa$(1) \vee$ $\neg$in $\wedge$ pa$(2) \vee \neg$in $\wedge$ pa(4)	0.1

For instance, the row on pa$(1, 2, 3)$ says that for it to be true, the package must initially be at 1 and cannot be inside the truck (otherwise, it would be moved along with the truck), and the computation took 0.05 seconds. The row on pa$(3, 2, 3)$ says that for it to be true, either the package was initially at 3 or it was inside the truck. The outputs for larger Ds are similar. The performance varies for different queries. For simple queries like ta$(1, 2, 3)$, their covers can be computed almost in constant time. The hardest one is for pa$(3, 2, 3)$ which took 25 minutes when $|D| = 7$.

It is interesting to compare our system with an alternative for computing the cover of a query. As we mentioned in Section 3, the set of abductive explanations according to Kakas and Mancarella is actually a cover. One way of computing these abductive explanations is to add, for each proposition $p \in A$, the following two clauses ([33]): $p \leftarrow \mathsf{not}\, \neg p$ and $\neg p \leftarrow \mathsf{not}\, p$ into the original program, and use the fact that there will be a one to one correspondence between abductive explanations of q under the original program and answer

[6]On a PIII 1GHz PC with 512MB RAM running SWI-Prolog 3.2.9.

sets of the new program that contain q. So one can use an answer set generator, for example **smodels** [35] or **dlv** [20] to compute a cover of query by generating all the answer sets in the new program that contain the query. However, the problem here is that there are too many such answer sets in this case. For instance, suppose there are n locations, then the number of answer sets that contain any particular query is in the order of 2^{2n}, roughly one half of the number of complete hypotheses, even for a very simple query like $\text{ta}(1, 2, 3)$. We do not know at the moment if there is any efficient way of using an answer set generator to compute a cover set of a query.

8 Final Remarks

Without the minimality requirement, a sound and complete procedure for abduction is required to generate sometimes a large amount of essentially redundant explanations. In this paper, we have given a new definition of abduction for logic programming that resolves this problem. In practice, for efficiency reasons one need not always compute the set of minimal explanations, but a cover, which may be considered a semantically adequate representation of all explanations.

Computationally, we have shown that explanations can be computed by confluent and terminating rewriting. On the one hand, our work explores the well-understood relationships among the completion semantics, the partial stable model semantics, and the answer set semantics. On the other hand, we combine several ideas that had only been studied previously in separate contexts. Namely, we build loop checking into rewrite systems that implement the completion semantics to obtain an abductive procedure for the partial stable model semantics.

There are several directions for extending this work. One of them is to consider rewriting for non-ground programs for some restricted yet decidable classes of non-ground goals. This would extend our rewriting procedure to a more general query-answering procedure for wider classes of applications. Another question is on the handling of constraints of the form:

$$\perp \leftarrow a_1, ..., a_i, \mathsf{not}\, b_1, ..., \mathsf{not}\, b_n.$$

Our new definition of abduction can be extended to include these constraints straightforwardly. Computationally, constraints may be handled in our rewriting procedure just like in other abductive procedures [4, 16, 9]: a goal is proved along with all the constraints. This ensures that all of the constraints are satisfied when the goal is proved. This approach actually produces a new semantics: partial stable model semantics with constraints. This is distinguished from the partial stable model semantics because a normal program under this new semantics is no longer guaranteed a partial stable model. The semantical implications of partial stable models with constraints worth further study.

To improve the efficiency of the goal rewriting procedure, space pruning techniques shall be investigated and incorporated. Scalability may be improved by considering different strategies of maintaining a goal so that the run time space usage can be reduced. For example, one possible strategy is not to expand a goal using the distribution rules SR6 and SR6'; instead, a collection of literals from the goal is selected, one at a time, that corresponds to a candidate solution. This requires book keeping mechanisms that should be closely related to maintaining backtrack points in implementing Prolog languages.

9 Acknowledgements

We would like to thank Ilkka Niemelä for helpful discussions related to the topics of this paper, and Ken Satoh for comments on an earlier version of this paper. The first author's work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6145/98E.

References

[1] R. Bol, A. Krzysztof, and J. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–39, 1991.

[2] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[3] L. Console, D. Theseider, and P. Porasso. On the relationship between abduction and deduction. *J. Logic Programming*, 2(5):661–690, 1991.

[4] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. *J. Logic Programming*, 34(2):111–167, 1998.

[5] N. Dershowitz and P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990.

[6] P. Dung. An argumentation theoretic foundation for logic programming. *J. Logic Programming*, 22:151–177, 1995.

[7] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In *Proc. 6th Int'l Conference on Logic Programming*, pages 234–254. MIT Press, 1989.

[8] F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[9] T. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–164, 1997.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th Int'l Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[11] M. Gelfond and V. Lifschitz. Classic negation in logic programs and disjunctive database. *New Generation Computing*, 9:365–385, 1991.

[12] L. Giordano, A. Martelli, and M. Sapino. Extending negation as failure by abduction: A three-valued stable model semantics. *J. Logic Programming*, 26(1):31–67, 1996.

[13] G. Huet. Confluent reductions: abstract perperties and applications to term rewriting systems. *JACM*, 27(4):797–821, 1980.

[14] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University, 1995.

[15] A. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conference on Artificial Intelligence*, pages 285–291, 1990.

[16] A. Kakas, A. Michael, and C. Mourlas. ACLP: abductive constraint logic programming. *J. Logic Programming*, 44(1-3):129–178, 2000.

[17] S. Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff publishing, 1971.

[18] K. Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992.

[19] K. Kunen. Signed data dependencies in logic programs. *J. Logic Programming*, 7(3):231–245, 1989.

[20] N. Leone et al. DLV: a disjunctive datalog system, release 2000-10-15. Vienna University of Technology, 2000.

[21] V. Lifschitz. Answer set programming. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 357–371. Springer, 1999.

[22] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1993. Morgan Kaufmann Publishers, 1995.

[23] F. Lin. From causal theories to successor state axioms: bridging the gap between nonmonotonic action theories and STRIPS-like formalisms. In *Proc. AAAI 2000*, pages 781–793, 2000.

[24] F. Lin and K. Wang. From causal theories to logic programs (sometimes). In *Proc. 5th LPNMR*. El Paso, Texas, Dec. 1999, 1999.

[25] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[26] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[27] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annual of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[28] David Poole. Representing knowledge for logic-based diagnosis. In *Proc. Fifth Generation Computer Systems Conference*, pages 1282–1290, 1988.

[29] T.C. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In *Proc. 7th Int'l Conference on Logic Programming*, pages 459–477. MIT Press, 1990.

[30] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *Proc. AAAI'87*, pages 183–189, 1987.

[31] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–217, 1990.

[32] T. Sato. Completed logic programs and their consistency. *J. Logic Programming*, 9(1):33–44, 1990.

[33] K. Satoh and N. Iwayama. Computing abduction using the TMS. In *Proc. 8th Int'l Conference on Logic Programming*, pages 505–518, 1991.

[34] K. Satoh and R. Iwayama. A query evaluation method for abductive logic programming. In *Proc. Joint Int'l Conference and Symposium on Logic Programming*, pages 671–685, 1992.

[35] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*. To appear.

[36] J. You and L. Yuan. A three-valued semantics for deductive databases and logic programs. *J. Computer and System Sciences*, 49:334–361, 1994.

[37] J. You and L. Yuan. On the equivalence of semantics for normal logic programs. *J. Logic Programming*, 22:212–221, 1995.

10 Appendix

We prove that a goal rewrite system with a finite program is confluent. It is known in the literature [13] that, a terminating rewrite relation is confluent iff it is *locally confluent*. Local confluence is defined as: whenever $t_1 \rightarrow t_2$ and $t_1 \rightarrow t_3$, there exist t_4 and rewrite sequences such that $t_2 \rightarrow^* t_4$ and $t_3 \rightarrow^* t_4$. It therefore suffices to show the property of local confluence.

Theorem 10.1 *Any goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ with a finite P is locally confluent.*

We introduce some notations.

A goal formula is viewed as a tree. A subformula is identified by a sequence of positive integers describing the path from the root symbol to the head of the subformula. These sequences are called *indices*. That an index ω identifies a subformula Φ is expressed by a mapping $m(\omega) = \Phi$. The empty sequence identifies the formula itself. For example, given $(l_1 \vee l_2) \wedge l_3$, we have $m(1) = l_1 \vee l_2$, and $m(1.2) = l_2$.

A rewrite sequence of zero or more steps is denoted as $Q_0 \rightarrow^* Q_k$. When we are interested in where in a given goal Q_i a rewrite occurs and which rule is applied, we write $Q_i \rightarrow_{\omega,r} Q_{i+1}$ to indicate that rule r is applied to the subformula at index ω.

Proof: Let $Q_0 \rightarrow^* Q_k$ be a rewrite sequence, where $k \geq 0$. Suppose $Q_k \rightarrow_{\omega,r} N$ and $Q_k \rightarrow_{\omega',r'} N'$. We consider all the cases of possibly different rewrites on Q_k.

If the two rewrite steps are independent of each other, i.e., if their indices are non-overlapping, then trivially, there exists a formula M such that $N \to_{\omega',r'} M$ and $N' \to_{\omega,r} M$.

The following are the overlapping cases.

Case 1. A loop rewrite at $m(\omega) = l$ and a rewrite by SR5 at $m(\omega') = T(C) \wedge l$ (the symmetric case of a rewrite by SR5' is similar). Using SR5 followed by SR2, we have $T(C) \wedge l \to F \wedge l \to F$. That is, an F is generated at ω'. A loop rule produces either an F, in which case we have $T(C) \wedge F \to F$ so that an F is at ω', or $T(C')$ for some C' at ω. The latter leads to $T(C) \wedge T(C') \to F$ due to $l \in C'$ and $\neg l \in C$, so that at ω' is also an F.

Case 2. Literal rewriting at $m(\omega) = l$ and a rewrite by SR5 at $m(\omega') = T(C) \wedge l$ (the symmetric case of a rewrite by SR5' is similar). Again SR5 leads to an F at ω'. Since P is finite, it follows from Proposition 4.7 that the sequence terminates at either an F, or a $T(C_1) \vee \ldots \vee T(C_m)$ for some $m \geq 1$. Hence, there exists an extension from $Q_k \to_{\omega,r} N$, say $Q_k \to_{\omega,r} M \to^* M'$ such that M' is the same as M except at ω, $m(\omega) = F$ or $m(\omega) = T(C_1) \vee \ldots \vee T(C_m)$, for some $m \geq 1$. That is, at ω' we have either $T(C) \wedge F$, or $T(C) \wedge (T(C_1) \vee \ldots \vee T(C_m))$ where $\neg l \in C$ and $l \in C_i$ for each $1 \leq i \leq m$. Clearly, in either case, the rewrite sequence can be extended to lead to an F at ω'.

Case 3. Any rewrite inside a subformula that is distributed by SR6 (the symmetric case is similar). SR6 is $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \to (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$. Any rewrite at a subformula inside $\Phi_1 \wedge (\Phi_2 \vee \Phi_3)$ causes overlapping. Clearly, distribution after the rewrite and the rewrite (or the duplicated rewrites if inside Φ_1) after distribution lead to the same result. Note that distribution does not change a rewrite chain for any literal; it simply duplicates the rewrite chain for each occurrence of the same literal.

Case 4. A rule overlaps with its symmetric counterpart. This includes the following cases of a rule and its symmetric counterpart both being applicable: SR1 and SR1' for goal $F \vee F$, SR2 and SR2' for goal $F \wedge F$, and SR6 and SR6' for goal $\Psi_1 \wedge \Psi_2$ where each Ψ_i is a disjunction. Clearly, in each case, there exist rewrite sequences leading to a common goal. ■