Recycling Computed Answers in Rewrite Systems for Abduction

Fangzhen Lin
Hong Kong University of Science and Technology
and
Jia-Huai You
University of Alberta

In rule-based systems, goal-oriented computations correspond naturally to the possible ways that an observation may be explained. In some applications, we need to compute explanations for a series of observations with the same domain. The question whether previously computed answers can be recycled arises. A yes answer could result in substantial savings of repeated computations. For systems based on classical logic, the answer is *yes*. For nonmonotonic systems however, one tends to believe that the answer should be *no*, since recycling is a form of adding information. In this paper, we show that computed answers can always be recycled, in a nontrivial way, for the class of rewrite procedures proposed earlier by the authors for logic programs with negation. We present some experimental results on an encoding of the logistics domain.

Categories and Subject Descriptors: I.2.3 [**ARTIFICIAL INTELLIGENCE**]: Deduction and Theorem Proving—*Logic programming*; *Nonmonotonic reasoning*

General Terms: Logic Programming, Abduction

Additional Key Words and Phrases: Stable model semantics, partial stable model semantics, rewrite systems, recycling

1. INTRODUCTION

The question we shall address in this paper is the following. With a sound and complete procedure for abduction, suppose we have computed explanations (conveniently represented as a disjunction) $Es = E_1 \vee ... \vee E_n$ for observation q. Suppose also that in the course of computing explanations for another observation p, we run into q again. Now, we may use the proofs Es for q without actually proving q again. The question is this: will the use (recycling) of the proofs Es for q in the proof for p preserve the soundness and completeness of the procedure?

In this paper, we answer this question positively, but in a nontrivial way, for the class of rewrite procedures proposed in [Lin and You 2001] for abduction in logic programming

An extended abstract of parts of this paper appeared in the proceedings of IJCAI-03, Acapulco, Mexico.
Author's address: F. Lin, flin@cs.ust.hk, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong.
Jia-Huai You, you@cs.ualberta.ca, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8.

under the (partial) stable model semantics ([Gelfond and Lifschitz 1988], [Przymusinski 1990]). The main result is a theorem (Theorem 4.7) that says recycling preserves the soundness and completeness.

The general idea of recycling is not new. Recycling in systems based on classical logic is always possible, since inferences in these systems can be viewed as transforming a logic theory to a logically equivalent one. In dynamic programming, it is the use of the answers for previously computed subgoals that reduces the computational complexity. In some game playing programs, for example in the world champion checker program *Shinook* (www.cs.ualberta.ca/˜chinook), the endgame database stores the computed results for endgame situations which can be referenced in real-time efficiently.

However, the problem of recycling in a nonmonotonic proof system has rarely been investigated. We note that recycling is to use previous proofs. This differs from adding consequences. For example, it is known that the semantics based on answer sets or (maximal) partial stable models [Dung 1991] do not possess the *cautious nonmonotonicity* property. That is, adding a consequence of a program could gain additional models thus losing some consequences. The following example is due to [Dix 1991]:

$$P = \{a \leftarrow \mathsf{not}\, b. \quad b \leftarrow c, \mathsf{not}\, a. \quad c \leftarrow a.\} \tag{1}$$

P has only one answer set, $\{a, c\}$. Thus, c is a consequence. When augmented with the rule $c \leftarrow$, the program gains a second answer set, $\{b, c\}$, and loses a as a consequence.

Abduction in the framework of logic programming has been studied extensively, and a number of formalisms and top-down query answering procedures have been proposed [Console et al. 1991; Dung 1991; 1995; Eshghi and Kowalski 1989; Kakas and Mancarella 1990a; Kakas et al. 1995; Kakas and Mancarella 1990b; Kakas et al. 2000; Lin and You 2001; Satoh and Iwayama 1991; 1992]. The class of rewrite procedures for abduction proposed in [Lin and You 2001] is based on the idea of *abduction as confluent and terminating rewriting*. These systems are called *canonical systems* in the literature of rewrite systems [Dershowitz and Jouannaud 1990]. The confluence and termination properties guarantee that rewriting terminates at a unique normal form independent of the order of rewriting. Thus, each particular strategy of rewriting yields a rewrite procedure. It has been shown in [Lin and You 2001] that these rewrite procedures are sound and complete under the semantics based on partial stable models for *brave reasoning*; i.e., a query is true in a partial stable model if and only there is a proof by such a rewrite procedure. Brave reasoning is particularly well suited for the task of generating explanations in abduction. In this paper, we show that the soundness and completeness of these rewrite procedures can be preserved when they are extended with rules that "recycle" previous results.

This paper is organized as follows. The next section defines logic program semantics. Section 3 reviews the rewriting framework. Then in Section 4 we formulate rewrite systems with computed rules and prove that recycling preserves soundness and completeness. Section 5 extends this result to rewrite systems with abduction. Recycling incurs overhead and sometimes the overhead could be substantial. A good recycling strategy makes computed answers likely to be used in later computations. In Section 6 we propose such a strategy, and Section 7 reports some experimental results.

2. LOGIC PROGRAM SEMANTICS

A rule is of the form

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

where a, b_i and c_i are atoms of the underlying propositional language $\mathcal{L}$. $\mathsf{not}\, c_i$ are called *default negations*. A *literal* is an atom ϕ or its negation $\neg\phi$. A *(normal) program* is a finite set of rules.

The *completion* of a program P, denoted $Comp(P)$, is a set of equivalences: for each atom $\phi \in \mathcal{L}$, if ϕ does not appear as the head of any rule in P, $\phi \leftrightarrow F \in Comp(P)$; otherwise, $\phi \leftrightarrow B_1 \vee ... \vee B_n \in Comp(P)$ (with default negations replaced by the corresponding negative literals) if there are exactly n rules $\phi \leftarrow B_i \in P$ with ϕ as the head; and these are the only equivalences in $Comp(P)$. We write T for B_i if B_i is empty.

The rewriting system of [Lin and You 2001] is sound and complete w.r.t. the partial stable model semantics [Przymusinski 1990]. A simple way to define partial stable models without even introducing 3-valued logic is by the so called *alternating fixpoints* [You and Yuan 1995]. Let P be a program and S a set of default negations. Define a function over sets S of default negations: $F_P(S) = \{\mathsf{not}\, a \mid P \cup S \not\vdash a\}$. The relation $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\, \phi$ being treated as a named atom not_ϕ.

A *partial stable model* M is defined by a fixpoint of the function that applies F_P twice, $F_P^2(S) = S$, while satisfying $S \subseteq F_P(S)$, in the following way: for any atom ξ, $\neg\xi \in M$ if $\mathsf{not}\, \xi \in S$, $\xi \in M$ if $P \cup S \vdash \xi$, and ξ is *undefined* otherwise. An *answer set* E (also called *stable model*) is defined by a fixpoint S such that $F_P(S) = S$ and $E = \{\xi \in \mathcal{L} \mid P \cup S \vdash \xi\}$.

3. GOAL REWRITE SYSTEMS

In this section, we briefly review goal rewrite systems as formulated in [Lin and You 2001].

A goal rewrite system is a rewrite system that consists of three types of rewrite rules: (1) Program rules from $Comp(P)$ for literal rewriting; (2) Simplification rules to transform and simplify goals; and (3) Loop rules for handling loops.

A *program rule* is a completed definition $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ used from left to right: ϕ can be rewritten to $B_1 \vee \ldots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \ldots \wedge \neg B_n$. These are called *literal rewriting*.

A *goal*, also called a *goal formula*, is a formula which may involve $\neg$, $\vee$ and $\wedge$. A goal resulted from a literal rewriting from another goal is called a *derived goal*. Like a formula, a goal may be transformed to another goal without changing its semantics. This is carried out by simplification rules.

We assume that in all goals negation appears only in front of a literal. This can be achieved by simple transformations like those used for converting a formula to clauses.

3.1 Simplification rules

The simplification rules constitute a nondeterministic transformation system formulated with a mechanism of loop handling in mind, which requires keeping track of literal sequences $g_0, \ldots, g_n$ where each g_i, $0 < i \leq n$, is in the goal formula resulted from rewriting g_{i-1}. Two central mechanisms in formalizing goal rewrite systems are *rewrite chains* and *contexts*.

—*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 the derived goal is generated in order to prove l. This ancestor-descendant relation is denoted $l \prec l'$. A sequence $l_1 \prec \ldots \prec l_n$ is then called a *rewrite chain*, abbreviated as $l_1 \prec^+ l_n$.

—*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*. A context will also be used to maintain consistency: if g can be proved via a conjunction, all of the conjuncts need be proved with contexts that are non-conflicting with each other. 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 any literal in a derived goal, the rewrite chain leading to it from a literal in the given goal is uniquely determined. As an example, suppose the completion of a program has the definitions: $a \leftrightarrow \neg b \wedge \neg c$ and $b \leftrightarrow q \vee \neg p$. Then, we get a rewrite sequence,

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

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

$$a \prec \neg b \prec \neg q$$
$$a \prec \neg b \prec p$$
$$a \prec \neg c$$

Simplification Rules: Let Φ and Φ_i be goal formulas, C be 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. $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$

SR5'. $(\Phi_1 \vee \Phi_2) \wedge \Phi_3 \rightarrow (\Phi_1 \wedge \Phi_3) \vee (\Phi_2 \wedge \Phi_3)$ □

For any goal formula, repeated applications of SR5 and SR5' transform it to a disjunctive normal form (DNF).

3.2 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. Two rewrite rules are formulated to handle loops.

Definition 3.1. 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*.

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. □

3.3 Goal rewrite systems

A *rewrite sequence* is a sequence of zero or more rewrite steps $Q_0 \rightarrow \ldots \rightarrow Q_k$, denoted $Q_0 \rightarrow^* Q_k$, 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 a subformula of Q_i, or
—applying a loop rule to a loop literal in Q_i.

Definition 3.2. A *goal rewrite system* for a program P is a triple $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$, 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 $\rightarrow$ is the set of all rewrite sequences.

In [Lin and You 2001], it is shown that goal rewrite systems are sound and complete w.r.t. the partial stable model semantics:

THEOREM 3.3. *Let P be a normal 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 for some $i \in [1..m]$, $g \in C_i \subseteq M$.*

4. GOAL REWRITE SYSTEMS WITH COMPUTED RULES

We now consider how to recycle computation in these rewrite systems. Notice that what we eventually want is a theory of recycling in abductive goal rewrite systems. But it is best to study the basic case where there are no abducibles first. As we shall see, just as abducible rewrite systems are a direct extension of rewrite systems as we showed in [Lin and You 2001], recycling in abducible rewrite systems is a direct extension of recycling in rewrite systems.

We first illustrate recycling with two examples.

EXAMPLE 4.1. *Given a rewrite system R^0, suppose we have a rewrite sequence $\neg q \rightarrow a \rightarrow a \rightarrow F$. The failure is due to a positive loop on a. We may recycle the computed* *answer by replacing the rewrite rule for $\neg q$ by the new rule, $\neg q \rightarrow F$. We thus get a new system, say R^1. Suppose in trying to prove g we have*

$$g \rightarrow a \rightarrow \neg q \rightarrow F$$

where the last step makes use of the computed answer for $\neg q$. The question arises as whether this way of using previously computed results guarantees the soundness and completeness. Theorem 4.7 to be proved later in this paper answers this question positively. To see it for this example, assume we have the following, successful proof in R^0

$$g \rightarrow a \rightarrow \neg q \rightarrow a \rightarrow T(\{g, a, \neg q\})$$

where the termination is due the even loop on a. Had such a sequence existed, recycling would have produced a wrong result. However, one can see that the existence of the rewrite step $a \rightarrow \neg q$ implies the existence of a different way to prove $\neg q$:

$$\neg q \rightarrow a \rightarrow \neg q \vee \ldots \rightarrow T(\{\neg q, a\}) \vee \ldots$$

contradicting that $\neg q$ was rewritten to F in R^0. □

Before giving the next example, we introduce a different way to understand rewrite sequences. Since any goal formula can always be transformed to a DNF using the distributive rules SR5 and SR5', and the order of rewriting does not matter, we can view rewriting as generating a sequence of DNFs. Thus, a rewrite sequence in DNF from an initial goal g,

$$g \rightarrow^* N_1 \vee ... \vee N_n$$

can be conveniently represented by *derivation trees*, or *d-trees*, one for each N_i representing one possible way of proving g. For any i, the d-tree for N_i has g as its root node, wherein a branch from g to a leaf node corresponds to a rewrite chain from g that eventually ends with an F or some $T(C)$. As N_i is a conjunction, a successful proof requires each branch to succeed (i.e. ends with $T(C)$ for some C) and the union of all resulting contexts to be consistent.

The next example is carefully constructed to illustrate that recycling may not yield the same answers as if no recycling were carried out. In particular, one can sometimes get additional answers.

EXAMPLE 4.2. *Consider the program:*

$$g \leftarrow a. \quad a \leftarrow \mathsf{not}\, b. \quad a \leftarrow e.$$
$$b \leftarrow b. \quad e \leftarrow p. \quad p \leftarrow a.$$

In Fig. 1, each d-tree consists of a single branch. The left two d-trees are expanded from goal p, corresponding to the following rewrite sequence:

$$p \rightarrow a \rightarrow e \vee \neg b \rightarrow p \vee \neg b \rightarrow F \vee \neg b \rightarrow \neg b \rightarrow \neg b \rightarrow T(\{p, a, \neg b\})$$

The next two d-trees are generated for goal g, corresponding to the rewrite sequence:

$$g \rightarrow a \rightarrow e \vee \neg b \rightarrow p \vee \neg b \rightarrow a \vee \neg b \rightarrow F \vee \neg b \rightarrow \neg b \rightarrow \neg b \rightarrow T(\{g, a, \neg b\})$$

Now, we recycle the proof for p in the proof for g and compare it with the one without recycling. Clearly, the successful d-tree for g (the fourth from the left) will still succeed

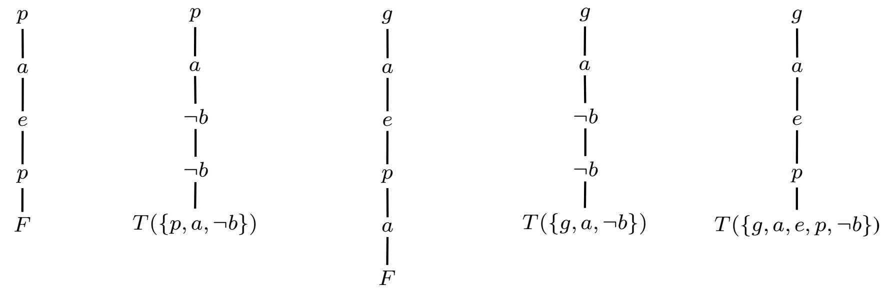

Fig. 1. Recycling may generate extra proofs

as it doesn't involve any p. The focus is then on the d-tree in the middle, in particular, the node p in it; this d-tree fails when no recycling was performed.

Since p is previously proved with context $\{g, a, \neg b\}$, recycling of this proof amounts to terminating p with a context which is the union of this context with the rewrite chain leading to p (see the d-tree on the right). But this results in a successful proof that fails without recycling.

Though recycling appears to have generated a wrong result, one can verify that both generated contexts, $\{g, a, \neg b\}$ and $\{g, a, e, p, \neg b\}$, belong to the same partial stable model. Thus, recycling in this example didn't lead to an incorrect answer but generated a redundant one. Theorem 4.7 shows that this is not incidental. Indeed, if p is true in a partial stable model, by derivation (look at the d-tree in the middle), so must be e, a, and g. □

4.1 Rewrite systems with computed rules

Given a goal rewrite system R, we may denote a rewrite sequence from a literal g by $g \rightarrow_R E$.

Definition 4.3. (Computed rule)
Let R be a goal rewrite system in which literal p is rewritten to its normal form. The *computed rule for* p is defined as: If $p \rightarrow_R F$, the computed rule for p is the rewrite rule $p \rightarrow F$; if $p \rightarrow_R T(C_1) \vee ... \vee T(C_n)$, then the computed rule for p is the rewrite rule $p \rightarrow T(C_1) \vee ... \vee T(C_n)$.

For the purpose of recycling, a computed rule $p \rightarrow E$ is meant to replace the existing literal rewrite rule for p. If E is F, i.e. the goal p failed, then it can be used directly as the literal rewrite rule for p. Otherwise, we must combine the contexts in E with the rewrite chain leading to p, and keep only consistent ones.

Recycling Rule:
Let $g_1 \prec^+ g_n$ be a rewrite chain where g_n is a non-loop literal. Let $G = \{g_1, ..., g_n\}$, and $g_n \rightarrow T(D_1) \vee ... \vee T(D_k)$ be the computed rule for g_n. Further, let $\{D'_1, ..., D'_{k'}\}$ be the subset of $\{D_1, ..., D_k\}$ containing any D_i such that $D_i \cup G$ is consistent. Then, the *recycling rule* for g_n is defined as:

RC. $g_n \rightarrow T(G \cup D'_1) \vee ... \vee T(G \cup D'_{k'})$ □

EXAMPLE 4.4. *Consider the following program:*

$$g \leftarrow a. \qquad a \leftarrow p. \qquad p \leftarrow \mathsf{not}\, a.$$
$$a \leftarrow \mathsf{not}\, p. \qquad p \leftarrow \mathsf{not}\, b. \qquad b \leftarrow \mathsf{not}\, a.$$

and the proof:

$$p \rightarrow \neg a \vee \neg b \rightarrow p \vee \neg b \rightarrow T(\{p, \neg a\}) \vee \neg b$$
$$\rightarrow T(\{p, \neg a\}) \vee p \rightarrow T(\{p, \neg a\}) \vee T(\{p, \neg b\})$$

We therefore have a computed rule for p*:*

$$p \rightarrow T(\{p, \neg a\}) \vee T(\{p, \neg b\})$$

Now, in the course of proving g *we can recycle the computed rule for* p*:*

$$g \rightarrow a \rightarrow p \vee \neg p \rightarrow T(\{g, a, p, \neg b\}) \vee \neg p \rightarrow ...$$

In the sequel, a rewrite system includes the recycling rule as well as zero or more computed rules. We note that the termination and confluence properties remain to hold for the extended systems.

We are interested in the soundness and completeness of a series of rewrite systems, each of which recycles computed answers generated on the previous one. For this purpose, given a program P we use R_P^0 to denote the original goal rewrite system where literal rewrite rules are defined by the Clark completion of P. For all $i \geq 0$, R_P^{i+1} is defined in terms of R_P^i as follows: Let Δ_i be the set of computed rules (generated) on R_P^i for the set of literals $\mathcal{L}_{\Delta_i}$ Then, R_P^{i+1} is the rewrite system obtained from R_P^i by replacing the rewrite rules for the literals in $\mathcal{L}_{\Delta_i}$ by those in Δ_i. In the rest of this section, we will always refer to a fixed program P. Thus we may drop the subscript P and write R^i.

Definition 4.5. A rewrite system R^i is *sound* iff, for any literal g and rewrite sequence $g \rightarrow_{R^i} T(C_1) \vee ... \vee T(C_n)$, and for each C_j, $j \in [1..n]$, there exists a partial stable model M of P such that $g \in C_j \subseteq M$. R^i is *complete* iff, for any literal g such that $g \in M$ for some partial stable model M of P, there is a rewrite sequence $g \rightarrow_{R^i} T(C_1) \vee ... \vee T(C_n)$ such that for some C_j, $j \in [1..n]$, $g \in C_j \subseteq M$.

An important property of provability by rewriting is the so-called *loop rotation*, which is needed in order to prove the soundness and completeness of recycling; namely, a proof (a successful branch in a d-tree) terminated by a loop rule can be captured in rotated forms.

To describe this property, we need the following notation about rewrite chains: Any direct dependency relation $l \prec l'$ may be denoted by $l \cdot l'$, and we allow a segment (which may be empty) of a rewrite chain to be denoted by a Greek letter such as δ, θ, and ξ. Thus, we may write $x \cdot \delta \cdot y$ to denote a rewrite chain from x to y via δ, and $x \cdot \delta$ to mean a rewrite chain that begins with x followed by the segment denoted by δ. A rewrite chain may also be used to denote the set of the literals on it.

LEMMA 4.6. *(Loop rotation)*
Let R^0 *be a rewrite system without computed rules. Let* Tr *be a d-tree for literal* g *that succeeds with context* C*. Suppose a branch of* Tr *ends with a loop,* $g \cdot \theta \cdot g$*, for some* θ*. Then, for any literal* $l \in \theta$*, there is a proof of* l *that succeeds with the same context* C*.*

Proof. A loop, $\pi = g \cdot l_1 \cdot l_2 \cdot \ldots \cdot l_n \cdot g$, where g and l_i are literals, can always be rotated as

$$l_1 \cdot l_2 \cdot \ldots \cdot l_n \cdot g \cdot l_1,$$
$$l_2 \cdot \ldots \cdot l_n \cdot g \cdot l_1 \cdot l_2,$$
$$\ldots\ldots$$

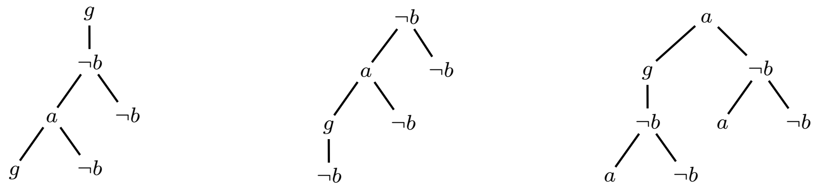

Fig. 2. Loop rotation

and so on, so that if π is a negative loop (or an even loop, resp.) so is its rotated loop. Rotation over a d-tree can be performed as follows: remove the top node n, and for any link from the top node, $n \cdot q$, attach the link $n \cdot q$ to any occurrence of n. The assumption of the existence of loop $g \cdot \theta \cdot g$ ensures that in every round of rotation there is at least one occurrence of the top node. (See Fig. 2 for an illustration where rotation proceeds from left to right.) It can be seen that the type of a loop is always preserved and the set of literals on the tree remains unchanged. □

4.2 Soundness and completeness of recycling

Below, given a literal l, by *a proof of l* we mean a rewrite sequence from l to $T(C_1) \vee \ldots \vee T(C_n)$, where any C_i can be referred to as a proof of l.

THEOREM 4.7. *For any $i \geq 0$, R^i is sound and complete.*

Proof. We prove the claim by induction on i. R^0, the system without computed rules, is sound and complete [Lin and You 2001]. Now assume for all j with $0 \leq j \leq i$, R^j are sound and complete, and show that R^{i+1} is also sound and complete.

We only need to consider the situations where rewriting in R^{i+1} differs from that of R^i. Let $\mathcal{L}_{\Delta_i}$ be the set of literals whose computed rules are generated on R^i. We can first carry out rewriting without rewriting the literals that are in $\mathcal{L}_{\Delta_i}$. In this case, rewriting from g in both R^i and R^{i+1} terminate at the same expression, which is either F or a DNF, say $N_1 \vee ... \vee N_m$. Each N_i can be represented by a d-tree.

Soundness: Suppose $g \rightarrow_{R^{i+1}} T(D_1) \vee ... \vee T(D_s)$. For any $D \in \{D_1, ..., D_s\}$ we need to show that there is a partial stable model M such that $D \subseteq M$. Consider the d-tree Tr that generates D and suppose g is its root node. We show inductively in a bottom-up fashion that all the literals on Tr must be in the same partial stable model. For any leaf node p that is terminated by its computed rule

$$p \rightarrow ... \vee T(C) \vee ...$$

suppose Tr is the one that succeeded with context C. By the inductive hypothesis on R^j, we know that R^j is sound for all $j \leq i$, thus there is a partial stable model M such that $C \subseteq D \subseteq M$. If a leaf node q is terminated by a loop, by the loop rotation lemma (lemma 4.1), there is a proof of q in R^i using rotated loops. Otherwise we have an obvious case where a leaf node is rewritten to $True$ by its Clark completion.

In the inductive step, let $l_1, ..., l_n$ be the child nodes of some node l and assume each l_i is proved in R^i hence in some partial stable model. We first show that they belong to the same partial stable model M. Then, we show that l can also be proved in R^i thus belonging to M as well. Without loss of generality, assume there are only two child nodes: $l_1 \rightarrow_{R^i} T(Q_1) \vee ... \vee T(Q_m)$, $l_2 \rightarrow_{R^i} T(W_1) \vee ... \vee T(W_n)$. Since D is constructed in

R^{i+1} using computed rules, by definitions of computed rule and the recycling rule, there are Q_i and W_j such that $Q_i \cup W_j \subseteq D$, and hence $Q_i \cup W_j$ is consistent. Then in R^i, the two contexts are merged by using simplification rule SR3, i.e.,

$$l_1 \wedge l_2 \rightarrow_{R^i} ... \vee [T(Q_i) \wedge T(W_j)] \vee ... \rightarrow_{R^i} ... \vee T(Q_i \cup W_j) \vee ...$$

Since R^i is sound, there is a partial stable model M such that $\{l_1, l_2\} \subseteq Q_i \cup W_j \subseteq M$. But l is derivable from l_1 and l_2. Using the definition of partial stable models, it can be shown that l must also be in M.

The induction allows us to conclude that for the top goal g and its proof D in R^{i+1}, we must have $g \in D \subseteq M$, for the same partial stable model M.

Completeness: We show that for any context generated in R^i, the same context will be generated in R^{i+1}. Then, R^{i+1} is complete simply because R^i is complete.

Let $p \in \mathcal{L}_{\Delta_i}$, and consider a proof of g via p and its d-tree. Since each branch of this d-tree can be expanded and eventually terminated independent of others, for simplicity, we consider a proof of g simply by (an extension of) a branch $g \cdot \xi \cdot p$. In R^{i+1} the computed rule for p is used while in R^i it is not. We only need to consider two cases of proof in R^i: either g is proved via p and a previously computed rule, or the proof is terminated due to a loop.

(i) The case of loops. In expanding the rewrite chain $g \cdot \xi \cdot p$ in R^i, we may form a loop, say $g \cdot \xi \cdot p \cdot \xi'$. If the loop is in ξ', exactly the same loop occurs in rewriting p as the top goal in R^i, so it is part of the computed rule for p. Otherwise it is a loop that *crosses over* p, in the general form

$$\pi = g \cdot \theta_1 \cdot l \cdot \theta_2 \cdot p \cdot \theta_3 \cdot l$$

where l is the loop literal. As a special case of loop rotation over a branch (cf. Lemma 4.1), the same way of terminating a rewrite chain presents itself in proving p as the top goal in R^i, which is

$$\pi' = p \cdot \theta_3 \cdot l \cdot \theta_2 \cdot p.$$

If the loop on π is a negative loop (or an even loop, resp.), so is π'. Thus the same context will be generated in R^{i+1}.

(ii) g is proved via p and a previously computed rule. That is, R^i gives a rewrite chain of the form $g \cdot \xi \cdot p \cdot \delta \cdot q$ where $q \rightarrow E$ is a computed rule generated on R^j for some $j < i$. Suppose the context generated this way is C. Because of the existence of $p \cdot \delta \cdot q$, exactly the same computed rule $q \rightarrow E$ must be used in generating the computed rule for p in R^i. It can be seen that the context generated in R^{i+1} by recycling the computed answers for p (which is computed via q) is exactly the same as the one that uses the computed answers for q but not those for p. So, for any context generated this way in R^i, the same context will be generated in R^{i+1} as well. □

As given in the corollary below, if we only recycle failed proofs then exactly the same contexts will be generated.

COROLLARY 4.8. *Let R^i be a rewrite system where each computed rule is of the form $p \rightarrow F$. Let g be a literal and E be a normal form. Then, for any $i \geq 0$, $g \rightarrow_{R^0} E$ iff $g \rightarrow_{R^i} E$.*

Proof. Let Δ be the set of literals whose rewrite rules are computed rules in R^i. By the condition in the corollary, each such computed rule is of the form $p \rightarrow F$. First consider the process of rewriting goal g to a normal form E in which each literal in Δ is treated as a terminal node. Note that E is a DNF, or a d-tree where some of the leaf nodes are literals in Δ. Then, continue rewriting from E in R^0 where each literal in Δ is eventually rewritten to F. Let the normal form be E'. Also continue rewriting from E in R^i where each literal in Δ is replaced by F, and thus the eventual normal form is exactly the same as E'. □

5. RECYCLING IN ABDUCTIVE REWRITE SYSTEMS

So we have shown that recycling can be done rewrite systems that does not have any abducibles. As we mentioned, this is just the base case. Our aim is to do recycling in rewrite systems with abducibles. Without abducibles, the basic problem is simply that, given a program P and a goal g, decide if there is a stable model that satisfies g. While the rewrite framework studied in this paper may turn out to be a good way of solving this problem, currently the best approach to answer this question is to use a stable model generator such as Smodels [Simons 2000] or ASSAT [Lin and Zhao 2004]. However, when there are many abducibles, and the problem is to find explanations of a goal in terms of these abducibles, then the rewrite systems here are better than answer set generators, as we discussed at the end of Section 7 in [Lin and You 2001]. In the following, we first review an abductive framework as given in [Lin and You 2001], we then review how rewrite systems are extended to handle abduction in [Lin and You 2001]. Finally, we show how our results about recycling in rewrite systems can be similarly extended to that in abducible rewrite systems.

5.1 An abductive logic programming framework

In the following, let P be a normal logic program, and A a set of atoms called *abducibles*. We also assume that abducibles do not occur as the head of a rule. All the definitions and results below are from [Lin and You 2001].

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.

Definition 5.1. A complete hypothesis α is said to be an explanation of an atom 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 5.2. 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 following logic program P

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

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.

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 5.3. 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 5.4. *If $\mathcal{S}$ is a cover of q, then each $\alpha \in \mathcal{S}$ must be an explanation of q.*

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:

PROPOSITION 5.5. *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha \in \mathcal{S}} \alpha$.*

Finally a remark here about the relationship between our notion of explanations and that of abductive explanations of Kakas and Mancarella's [Kakas and Mancarella 1990a]. It can be shown 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. So what is new here is a notion of minimal explanations. This is important as the number of complete explanations can be significantly larger than the number of minimal explanations. More importantly, minimal explanations capture a notion of "relevance" that is not there in complete explanations. More discussions about this can be found in [Lin and You 2001].

5.2 Abductive Rewrite Systems

Again, all results in this subsection are from [Lin and You 2001]. Firstly, as shown in [Lin and You 2001], the rewriting framework can be extended to 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 $SR2'$. In a similar way, the goal rewrite systems with computed rules in the previous section can be extended to abduction as well.

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 3.1), 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. Recall that $\mathcal{Q}_L$ is the set of all goals. The following theorem from [Lin and You 2001] shows that these rewrite systems are both sound and complete with respect to the partial stable models semantics.

THEOREM 5.6. *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 $g \in \bigcup_{1\leq j\leq k} C_j \subseteq M$.

Notice that our soundness and completeness result is stated with respect to partial stable models, but our abductive framework in section 5 is defined with the stable model semantics in mind. In general, a stable model is also a partial stable model, but the converse is not always true. Thus our rewrite systems may not be sound under the stable model semantics. The reason is that stable models check for global consistency, but our rewrite systems check only local ones. There are several ways to make the rewriting system also sound and complete for stable models. One way is that 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 [Fages 1994; Sato 1990].

The following theorem is proved in [Lin and You 2001].

THEOREM 5.7. *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*

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

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.

5.3 Recycling in Abductive Rewrite Systems

We now show that recycling can be extended to abductive rewrite systems as well.

Definition 5.8. (Computed rule for abduction)
Let R be a goal rewrite system for abduction. The *computed rule for a literal* p is defined as: If $p \rightarrow_R F$, the computed rule for p is the rewrite rule $p \rightarrow F$; if

$$p \rightarrow_R [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]$$

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, then the computed rule for p is the rewrite rule

$$p \rightarrow [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})] \tag{2}$$

Recycling Rule:
Let $g_1 \prec^+ g_n \prec p$ be a rewrite chain where p is a non-loop literal. Let $G = \{g_1, ..., g_n, p\}$, and (2) be the computed rule for p. Then, the *recycling rule* for p is defined as:

RC'.

$$p \rightarrow [l_{11}(C_{11} \cup G) \wedge \cdots \wedge l_{1k_1}(C_{1k_1} \cup G)] \vee \ldots \vee [l_{m1}(C_{m1} \cup G) \wedge \cdots \wedge l_{mk_m}(C_{mk_m} \cup G)]$$

Now given an abductive rewrite system $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$. Recursively, let R^n be the rewrite system obtained from R^{n-1} by replacing the rewrite rules for *some* of the literals by their corresponding recycling rules. Notice here that R^n is obtained from R^{n-1} nondeterministically by choosing some of the literals whose answers have been computed in R^{n-1}, and replace their rewrite rules by these computed answers. Both Theorems 5.6 and 5.7 can be extended to R^i for any $i \geq 0$.

THEOREM 5.9. *Let P be a finite program, A a set of abducibles, and $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the goal rewrite system with respect to P and A. For any $i > 0$, let R^i be defined as above.*

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

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

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*

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

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

Proof. Similar to the proof of Theorem 4.7. □

THEOREM 5.10. *Let P be a finite program, and $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ a goal rewrite system. Let R^i be as defined above, $i > 0$. Suppose q is a proposition and*

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

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. Just like the proof of Theorem 5.7 in [Lin and You 2001]. □

6. A RECYCLING STRATEGY

We have shown that in theory one can reuse the previously computed answers in our rewrite systems for abduction. To put the theory into practice, we need some effective strategies on how to recycle these computations.

First, we shall further motivate the need for good recycling strategies. Since a set of previously computed answers may not be actually used in answering a new goal, recycling in general does not change the complexity of deciding whether a goal is true in a partial stable model of a program. However, recycling sometimes can reduce the computation of a hard goal to a trivial one simply due to the fact that the real hard part of the current goal has already been computed and can be used directly in proving the current goal.

However, the benefit of recycling does not come free: recycling incurs overhead, sometimes substantial overhead. In the worse case, since there can be an exponential number of (partial) stable models for a program, there can be an exponential number of explanations for an observation. Saving all of them can be expensive. Furthermore, one needs to check against the list of computed literals during computation. So, it is easy to construct an example where computation becomes less efficient due to overhead. Thus, the effectiveness of recycling depends on a good strategy that previously computed results are likely to be used. In this section we present such a strategy.

If we want to compute the abduction of all goals in a set, without the framework of recycling introduced here, the only way is to compute them one by one independently. With the idea of recycling, we can try to recycle previously computed answers. The question is then which goals to compute first. This question arises even if we just want to compute the abduction of a single goal: instead of computing it using the original program, it may sometimes be better if we first compute the abduction of some other goals and recycle the results.

Assuming that goals are literals, a simple strategy for deciding the order of goals to be computed is to find out the dependency relations among the goals.

Definition 6.1. A literal l is said to be depending on a literal l' if the atom in l depends on the atom in l'. An atom p is said to be depending on an atom q if either q is in the body of a rule whose head is p or inductively, there is another atom r such that p depends on r, and q is in the body of a rule whose head is r.

Suppose l' depends on l, but l does not depend on l', written $l < l'$ below. It is easy to see that l will never be sub-goaled to l' during rewriting, but l' could be sub-goaled to l. Thus if we need to compute the abduction of both l and l', we should do it for l first.

This strategy, and as we have mentioned above in general rewrite with computed rules, is not guaranteed to be more efficient computationally:

—$l < l'$ does not imply that a proof of l' must go through l. For instance, it could be that all rewrite chains for l' terminate with failure before l is reached.

—Even if we have a computed rule for l, using it in the proof of l' does not always result in computational gains. For instance, suppose the computed rule for l is

$$l \rightarrow_R T(C_1) \vee \cdots \vee T(C_n),$$

and a rewrite chain for l' is $l \rightarrow^* T(C) \wedge l$. If C is inconsistent with all C_i's, then this rewrite chain will terminate with F after trying out all n different proofs of l using the computed rule for l. However, if we do not use the computed rule for l, it may well be that after only one step of expanding l, a contradiction is detected.

However, there is one case of recycling that is almost trivial to implement, yet its effectiveness is independent of any strategy of rewriting. This is the case of recycling only failed proofs. That is, if the computed rule for l is $l \rightarrow_R F$, then using this computed rule for l in the proof of another goal is always as good as the one without. More precisely, if a terminating rewrite chain of q goes through l but does not use the computed rule for l, then there is another terminating rewrite chain of q that uses the computed rule for l, and has the same or fewer number of steps than the original rewrite chain. It is impossible to quantify the computational gains in terms of worst-case complexity analysis. In practice, we expect it to be very effective as our following experiment shows.

7. EXPERIMENTS

We have implemented a depth-first search rewrite procedure with branch and bound. The procedure can be used to compute explanations using a nonground program, under the condition that in each rule a variable that appears in the body must also appear in the head. When this condition is not satisfied, one only needs to instantiate those variables that only appear in the body of a rule.

To check the effectiveness of the idea of recycling, we consider an application of abduction in logic programming, the problem of computing successor state axioms from a causal action theory [Lin 1995; Lin and Wang 1999; Lin 2003].

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;[1]

—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:[2]

$$\begin{aligned}
&\text{ta}(X, X1, X). &(3)\\
&\text{pa}(X, X1, X2) \leftarrow \text{ta}(X, X1, X2), \text{in}(X1, X2).\\
&\text{ta}(X, X1, X2) \leftarrow X \neq X2, \text{ta}(X), \textsf{not}\,\text{taol}(X, X1, X2).\\
&\text{taol}(X, X1, X2) \leftarrow Y \neq X, \text{ta}(Y, X1, X2).\\
&\text{pa}(X, X1, X2) \leftarrow \text{pa}(X), \textsf{not}\,\text{paol}(X, X1, X2).\\
&\text{paol}(X, X1, X2) \leftarrow Y \neq X, \text{pa}(Y, X1, X2).\\
&\text{in}(X, Y) \leftarrow \text{in}. &(4)
\end{aligned}$$

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

[1]Suppose $move(y, z)$ stands for the action of driving the truck from locations y to z, and the current situation is s, then in situation calculus, this proposition can be written as $holds(ta(x), do(move(y, z), s))$. Our representation is sufficient here since we only consider one type of actions, and two situations: the initial one, and the one after the execution of the action.

[2]See [Lin and Wang 1999] for how logic programs of this kind can be used to encode a class of causal action theories.

Query	9 locations		10 locations	
	NR	WR	NR	WR
pa(1,2,3)	0.71	0.41	1.50	0.89
-pa(1,2,3)	75.89	2.28	342.96	5.65
pa(3,2,3)	137.05	0.89	630.69	1.98
-pa(3,2,3)	2.97	1.98	7.64	5.03
pa(1,5,7)	122.87	0.75	278.07	1.31
-pa(1,5,7)	727.6	7.07	2534.09	19.08
pa(7,5,1)	108.66	17.82	188.50	30.72
-pa(7,5,1)	74.43	2.26	340.51	5.64
pa(7,1,7)	7619.72	20.78	29140.69	35.65
-pa(7,1,7)	2.98	2.01	7.71	5.05

Table I. Recycling in logistics domain. Legends: NR - no recycling; WR - recycling ta(X, Y, Z) goals. All times are in CPU seconds.

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:

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

Our job here is then to compute the abduction of successor state propositions

$$\{\text{ta}(x, y, z), \text{pa}(x, y, z), \text{in}(y, z)\}$$

in terms of these abducibles, which are initial state propositions. For instance, given query pa$(3, 2, 3)$, our system would compute its abduction as pa$(3) \vee$ in, meaning that for it to be true, either the package was initially at 3 or it was inside the truck.

According to the definition in the last section, literals that contain pa(X, Y, Z) depend on those that contain in(X, Y) and ta(X, Y, Z). But literals that contain in(X, Y) and those that contain ta(X, Y, Z) do not depend on each other. So we should compute first the abduction of in(X, Y) and ta(X, Y, Z). Now in(X, Y) is solved by rule (4), ta(X, Y, X) by rule (3), and as it turned out, when $X \neq Z$, ta(X, Y, Z) is always false, and its computation is relatively easy. For instance, in our experiments for the domain with 9 locations, query ta$(7, 1, 6)$ took only 2.6 seconds. In comparison, query pa$(7, 1, 7)$ took more than 7000 seconds without recycling.

Table I contains run time data for some representative queries.[3] For comparison purpose, each query is given two entries: the one under "NR" refers to regular rewriting system without using recycling, and the one under "WR" refers to rewriting system using computed rules about ta(X, Y, Z). As one can see, especially for hard queries like pa$(7, 1, 7)$, recycling in this case significantly speeds up the computation.

[3]Our implementation was written in Sicstus Prolog, and the experiments were done on a PIII 1GHz notebook with 512 MB memory. This implementation is a significant improvement over the one in [Lin and You 2001]. For instance, for a domain with 7 locations query pa$(3, 2, 3)$ took more than 20 minutes for the implementation reported in [Lin and You 2001], but required less than 1 second under our implementation running on a comparable machine.

8. EXTENSIONS AND RELATED WORK

8.1 Some Extensions

So far we have assumed that the given logic program is propositional, and does not contain constraints. We discussed in [Lin and You 2001] some possible extensions. Firstly, our rewrite systems work on a nonground program, under the condition that in each rule a variable that appears in the body must also appear in the head. Under this condition, an observation (a ground goal) is always rewritten to another ground goal, so that a rewriting mechanism designed for ground programs works just as well. When the condition is not satisfied, one only needs to instantiate those variables that only appear in the body of a rule. For example, domain restricted programs [Niemelä 1999] can be instantiated only on domain predicates for variables that do not appear in the head. This is a significant departure from the approaches that are based on ground computation where a function-free program is first instantiated to a ground program with which the intended models are then computed.

Constraints of the form

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

can be handled in our rewriting procedure just like in other abductive procedures [Denecker and Schreye 1998; Fung and Kowalski 1997; Kakas et al. 2000]: namely, a goal is proved along with all the constraints. This ensures that all of the constraints are satisfied when the goal is proved.

EXAMPLE 8.1. *Consider the following program and constraint:*

$$a \leftarrow \mathsf{not}\, b. \quad b \leftarrow \mathsf{not}\, a. \quad c \leftarrow c.$$
$$\perp \leftarrow b, \mathsf{not}\, c.$$

In trying to prove b, for example, the goal formula including the constraint is $b \wedge \neg(b \wedge \neg c)$, which reduces to $b \wedge (\neg b \vee c)$. The following rewrite sequence shows that there is no partial stable model that contains b while satisfying the given constraint.

$$\begin{aligned}
& b \wedge (\neg b \vee c) \\
& \rightarrow (b \wedge \neg b) \vee (b \wedge c) \rightarrow ...... \\
& \rightarrow F \vee (b \wedge c) \rightarrow b \wedge c \rightarrow \neg a \wedge c \rightarrow b \wedge c \\
& \rightarrow T(\{b, \neg a\}) \wedge c \rightarrow T(\{b, \neg a\}) \wedge c \rightarrow T(\{b, \neg a\}) \wedge F \rightarrow F
\end{aligned}$$

However, computationally this is not a very effective approach when there are many constraints. Since a rewrite chain in our system carries a context with it, we could check the consistency of this context with the given constraints. This can be done by adding the following simplification rule:

$$T(C) \rightarrow F, \text{ if } C \text{ violates a constraint}$$

Here we say that a context C violates a constraint $\perp \leftarrow G$ if G is a subset of C.

However, this approach may not be "sound", for the same reason that our rewrite system is only sound under partial stable model semantics, but not under the stable model semantics. Consider the following program:

$$p. \quad \perp \leftarrow p, q. \quad \perp \leftarrow p, \mathsf{not}\, q.$$

There is a rewrite of p to T under the context $\{p\}$, which does not violate any of the constraints according to our definition. However, p should not be "proved" as the context cannot be extended to one that satisfies both of the constraints. But this is the same reason why our rewrite system is not sound under the stable model semantics: a program may not have a stable model (for instance with $a \leftarrow \mathsf{not}\, a$ the only rule whose head is a), but a goal may be written to T (for instance for a goal that does not mention a). So perhaps this strategy of dealing with constraints matches well with our rewrite systems, which are sound and complete under the partial stable model semantics.

8.2 Related work

We discussed in [Lin and You 2001] how our rewrite systems for abduction are related to other abductive systems, e.g. [Console et al. 1991; Denecker and Schreye 1998; Eshghi and Kowalski 1989; Fung and Kowalski 1997; Kakas et al. 1995; Kakas and Mancarella 1990a; Kakas et al. 2000; Satoh and Iwayama 1992]. One useful way of looking at an abductive logic programming system is the underlying semantics the system is based on. For instance, The rewrite-like systems in [Console et al. 1991; Fung and Kowalski 1997; Hayashi 1998] are based on Clark's completion semantics, the systems in [Denecker and Schreye 1998; Kakas et al. 2001] are for well-founded models, and the procedure in [Eshghi and Kowalski 1989] is sound and complete under the finite-failure three-valued semantics in which loops causing infinite failure are modeled by the truth value *undefined* [Giordano et al. 1996].

Our rewrite systems were designed with the stable model semantics in mind. As we see in section 5, we defined basic notions of abduction in logic programming, such as explanations, minimal explanations, and covers of explanations in terms of the stable model semantics. However, our rewrite systems are goal-directed, thus check for local consistency only. For instance, unless the atom a is "related to" to the query q, a rewrite chain starting at q would never reach a, thus would never check the consistency implicitly given by the odd loop $a \leftarrow \mathsf{not}\, a$. Thus our rewrite systems are sound and complete only under the partial stable model semantics.

According to [Janhunen et al. 2004], given any normal or disjunctive program P, there is a polynomial time translation from P to P' such that a query is true in an answer set of P if and only if it is true in a partial stable model of P' (see Corollaries 3.15-16 of [Janhunen et al. 2004]). So, query answering under the answer set semantics can be carried out by query answering under the semantics based on partial stable models. Thus, our rewrite systems are applicable to the stable model semantics as well.

One can also apply the technique of rewriting for the answer set semantics without carrying out a transformation, as we outlined following Theorem 5.6.

We note that recycling in our case differs from the reuse of hypotheses in the abductive procedures in [Eshghi and Kowalski 1989; Kakas and Mancarella 1990b; Kakas et al. 2000]. In the latter case, hypotheses are saved as individuals for the purpose of checking consistency while in our case, the computed answers are saved. It is interesting to see that consistency check in our rewrite procedures is already captured by the loop detection mechanism as defined in Section 3. The advantage of loop detection is that not only the completeness is guaranteed but also the procedure is terminating for propositional programs.

9. CONCLUDING REMARKS AND FUTURE WORK

We have considered the problem of how to reuse previously computed results for answering other queries in the abductive rewriting system of Lin and You [Lin and You 2001] for logic programs with negation, and showed that this can indeed be done. We have also described a methodology of using the recycling system in practice by analyzing the dependency relationship among propositions in a logic programs. We applied this methodology to the problem of computing the effect of actions in a logistics domain, the same one considered in [Lin and You 2001], and our experimental results showed that recycling in this domain can indeed result in good performance gain.

For future work we are looking for more domains to try our system on and to implement a system that can automatically analyze a program and decide how best to recycle previous computations.

ACKNOWLEDGMENTS

We would like to thank the reviewers of this paper for their thoughtful comments. Fangzhen Lin's work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6205/02E.

REFERENCES

CONSOLE, L., THESEIDER, D., AND PORASSO, P. 1991. On the relationship between abduction and deduction. *J. Logic Programming 2(5)*, 661–690.

DENECKER, M. AND SCHREYE, D. D. 1998. SLDNFA: an abductive procedure for normal abductive programs. *J. Logic Programming 34,* 2, 111–167.

DERSHOWITZ, N. AND JOUANNAUD, P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics*. North-Holland, 243–320.

DIX, J. 1991. Classifying semantics of logic programs. In *Proc. First Workshop on Logic Programming and Nonmonotonic Reasoning*. MIT Press, 167–180.

DUNG, P. 1991. Negations as hypothesis: An abductive foundation for logic programming. In *Proc. 8th International Conference on Logic Programming*. MIT press, 3–17.

DUNG, P. 1995. An argumentation theoretic foundation for logic programming. *J. Logic Programming 22*, 151–177.

ESHGHI, K. AND KOWALSKI, R. 1989. Abduction compared with negation by failure. In *Proc. 6th International Conference on Logic Programming*. MIT Press, 234–254.

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

FUNG, T. AND KOWALSKI, R. 1997. The iff proof procedure for abductive logic programming. *J. Logic Programming 33,* 2, 151–164.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proc. 5th International Conference on Logic Programming*. MIT Press, 1070–1080.

GIORDANO, L., MARTELLI, A., AND SAPINO, M. 1996. Extending negation as failure by abduction: A three-valued stable model semantics. *J. Logic Programming 26,* 1, 31–67.

HAYASHI, H. 1998. Knowledge assimilation and proof restoration through the addition of goals. In *Proc. 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. LNCS 1480. 291–302.

JANHUNEN, T., NIEMELÄ, I., SEIPEL, D., SIMONS, P., AND YOU, J. 2004. Unfolding partiality and disjunctions in stable model semantics. In *ACM Transactions on Computational Logic*. To appear.

KAKAS, A., KOWALSKI, R., AND TONI, F. 1995. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University.

KAKAS, A. AND MANCARELLA, P. 1990a. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conference on Artificial Intelligence*. 285–291.

KAKAS, A. AND MANCARELLA, P. 1990b. On the relation between truth maintenance and abduction. In *Proc. PRICAI'90*. 438–443.

KAKAS, A., MICHAEL, A., AND MOURLAS, C. 2000. ACLP: abductive constraint logic programming. *J. Logic Programming 44,* 1-3, 129–178.

KAKAS, A., VAN NUFFELEN, B., AND DENECKER, M. 2001. A-system: problem solving through abduction. In *Proc. IJCAI'01*. 591–596.

LIN, F. 1995. Embracing causality in specifying the indirect effects of actions. In *Proc. IJCAI'95*. Morgan Kaufmann Publishers, 1985–1993.

LIN, F. 2003. Compiling causal theories to successor state axioms and STRIPS-like systems. *Journal of Artificial Intelligence Research 19*, 279–314.

LIN, F. AND WANG, K. 1999. From causal theories to logic programs (sometimes). In *Proc. 5th International Conference on Logic Programming and Nonmonotonic Reasoning, LNCS 1730*. 117–131.

LIN, F. AND YOU, J. 2001. Abduction in logic programming: a new definition and an abductive procedure based on rewriting. In *Proc. IJCAI'01*. 655–661. The full paper appears in *Artificial Intelligence* 140(1/2): 175-205 (2002).

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157(1-2)*, 115–137.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annual of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

PRZYMUSINSKI, T. 1990. Extended stable semantics for normal and disjunctive logic programs. In *Proc. 7th International Conference on Logic Programming*. MIT Press, 459–477.

SATO, T. 1990. Completed logic programs and their consistency. *J. Logic Programming 9,* 1, 33–44.

SATOH, K. AND IWAYAMA, N. 1991. Computing abduction using the TMS. In *Proc. 8th International Conference on Logic Programming*. MIT Press, 505–518.

SATOH, K. AND IWAYAMA, R. 1992. A query evaluation method for abductive logic programming. In *Proc. Joint International Conference and Symposium on Logic Programming*. MIT Press, 671–685.

SIMONS, P. 2000. Smodels: a system for computing the stable models of logic programs, version 2.26. At `http://www.tcs.hut.fi/Software/smodels/`.

YOU, J. AND YUAN, L. 1995. On the equivalence of semantics for normal logic programs. *J. Logic Programming 22*, 212–221.

Received February 2004; revised November 2004; accepted April 2005