Recycling Computed Answers in Rewrite Systems for Abduction

Fangzhen Lin*
http://www.cs.ust.hk/~ flin
Department of Computer Science
Hong Kong University of Science and Technolog
Clear Water Bay, Kowloon, Hong Kong

Jia-Huai You
http://www.cs.ualberta.ca/~ you
Department of Computer Science
University of Alberta
Edmonton, Alberta, Canada

Abstract

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 classic 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 in [Lin and You, 2001] for logic programs with negation. We present some experimental results on an encoding of a logistics domain.

1 Introduction

Logic programming with answer sets [Gelfond and Lifschitz, 1988] or partial stable models [Przymusinski, 1990] has been adopted as a framework for abduction, and a number of formalisms and top-down query answering procedures have been proposed [Dung, 1991; Eshghi and Kowalski, 1989; Kakas and Mancarella, 1990; Kakas *et al.*, 2000; Lin and You, 2001; Satoh and Iwayama, 1992].

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 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]. The main result is a theorem (Theorem 4.6) that says recycling preserves the soundness and completeness.

The class of rewrite procedures 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.

The general idea of recycling is not new. Recycling in systems based on classic 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 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.

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, and Section 6 reports some experimental results.

*This author's work was supported in part by HK RGC CERG HKUST6205/02E.

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. 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 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

We introduce 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 using the following rules: for any formulas Φ and Ψ,

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

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 rewrite chains from a: $a \prec \neg b \prec \neg q$; $a \prec \neg b \prec p$; and $a \prec \neg c$.

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

SR3 merges two contexts if they contain no complementary literals, otherwise SR4 makes it a failure to prove. Repeated applications of SR5 and SR5' can transform any goal formula 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. $g_n \to F$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a positive loop or an odd loop.

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

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 a subformula of Q_i, or
- applying a loop rule to a loop literal in Q_i.

Example 3.2 *For the program given in the Introduction, $P_0 = \{a \leftarrow \mathsf{not}\, b.\ \ b \leftarrow c, \mathsf{not}\, a.\ \ c \leftarrow a.\}$, a is proved but b is not. This is shown by the following rewrite sequences:*

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

Let $P_1 = \{b \leftarrow \mathsf{not}\, c.\ \ c \leftarrow c.\}$. b is proved and $\neg b$ is not.

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

3.3 Previous results

In [Lin and You, 2001], it is shown that (1) a goal rewrite system is a canonical system, i.e., it confluent and terminating; (2) any goal rewrite system is sound and complete w.r.t. the partial stable model semantics; and (3) the rewriting framework can be extended for abduction in a relatively straightforward manner.

4 Goal Rewrite Systems with Computed Rules

We first use two examples to illustrate the main technical results of this section.

Example 4.1 *Given a rewrite system R^0, suppose we have a rewrite sequence $\neg q \to a \to a \to F$. 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, $\neg q \to F$. We thus get a new system, say R^1. Suppose in trying to prove g we have*

$$g \to a \to \neg q \to F$$

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 4.6 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

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

where the termination is due the even loop on a. Had such a sequence existed, recycling would have produced a wrong

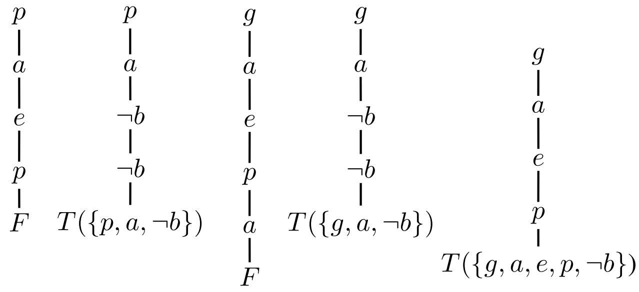

Figure 1: Recycling may generate extra proofs

result. However, one can see that the existence of the rewrite step $a \to \neg q$ implies the existence of a different way to prove $\neg q$:

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

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,

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

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 such a disjunct is a conjunction, a successful proof requires each branch to succeed 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:

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

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

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

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 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 4.6 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 a computed rule is of the form $p \rightarrow F$ representing a failed derivation, 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. $g_n \rightarrow T(G \cup D'_1) \vee ... \vee T(G \cup D'_{k'})$ □

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^0_P 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^{i+1}_P is defined in terms of R^i_P as follows: Let Δ_i be the set of computed rules (generated) on R^i_P for the set of literals $\mathcal{L}_{\Delta_i}$ Then, R^{i+1}_P is the rewrite system obtained from R^i_P by replacing the rewrite rules for the literals in $\mathcal{L}_{\Delta_i}$ by those in Δ_i. In the rest of this sectin, we will always refer to a fixed program P. Thus we may drop the subscript P and write R^i.

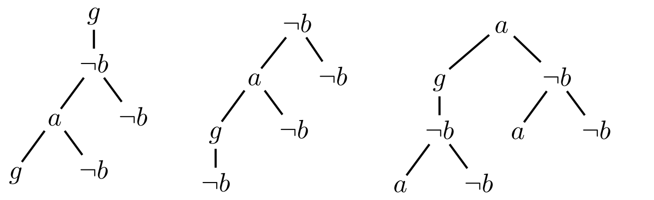

Figure 2: Loop rotation

Definition 4.4 *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 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, or $x \cdot \delta$ 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 4.5 *(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

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

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

Theorem 4.6 *For any $i \geq 0$, R^i is sound and complete.*

We sketch how this can be proved. We can 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.

For the soundness, we assume $g \rightarrow_{R^{i+1}} T(D_1) \vee ... \vee T(D_s)$. 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 can show inductively in a bottom-up fashion that all the literals on Tr belong to the same partial stable model.

For the completeness, we can 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 in R^i that goes through p. In particular, consider a d-tree of this proof that contains p. Since each branch of this d-tree can be expanded and eventually terminated independent of others, for simplicity, we consider how a branch $g \cdot \xi \cdot p$ in the d-tree is extended. 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 in R^i: either g is proved via p and a previously computed rule, or the branch is terminated due to a loop. Here, let us consider the latter only. 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}.

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

Corollary 4.7 *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 $g \rightarrow_{R^i} E$.*

5 Recycling in Abductive Rewrite Systems

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. Once a goal is rewritten to a disjunction of conjunctions of abducible literals and T:

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

where 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 this result can be recycled.

6 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. 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.

To check the effectiveness of the idea of recycling, we tested our system on the logistics problem in [Lin and You, 2001]. This is a domain in which there is a truck and a package. A package can be in or outside a truck, and a truck can be moved from one location to another. The problem is that given state constraints such as 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 does the package, how we can derive a complete specification of the effects of the action of moving a truck from one location to another. Suppose that we have the following propositions: ta(x) (pa(x)) – the truck (package) is at location x initially; in – the package is in the truck initially; ta(x, y, z) (pa(x, y, z)) – the truck (package) is at location x after performing the action of moving it from y to z; in(y, z) – the package is in the truck after performing the action of moving the truck from y to z. Then in [Lin and You, 2001], the problem is solved by computing the abduction of successor state propositions $\{\text{ta}(x, y, z), \text{pa}(x, y, z), \text{in}(y, z)\}$ in terms of initial state propositions $\{\text{ta}(x), \text{pa}(x), \text{in}\}$ (abducibles) using the following logic program (see [Lin and You, 2001] for more details):

$$\text{ta}(X, X1, X). \tag{2}$$

$$\text{pa}(X, X1, X2) \leftarrow \text{ta}(X, X1, X2), \text{in}(X1, X2). \tag{3}$$

$$\text{ta}(X, X1, X2) \leftarrow X \neq X2, \text{ta}(X), \text{not taol}(X, X1, X2). \tag{4}$$

$$\text{taol}(X, X1, X2) \leftarrow Y \neq X, \text{ta}(Y, X1, X2). \tag{5}$$

$$\text{pa}(X, X1, X2) \leftarrow \text{pa}(X), \text{not paol}(X, X1, X2). \tag{6}$$

$$\text{paol}(X, X1, X2) \leftarrow Y \neq X, \text{pa}(Y, X1, X2). \tag{7}$$

$$\text{in}(X, Y) \leftarrow \text{in}. \tag{8}$$

Here the variables are to be instantiated over a domain of locations. 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.

Which subgoals to compute first? If we want to compute the abduction of all propositions, 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 results. 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 result.

A simple solution is to find out the dependency relations among the propositions: a proposition p depends on q if either q is in the body of a rule whose head is p or inductively, there is another proposition r such that p depends on r, and r depends on q.

For our program above, it is easy to see that among the propositions that we want to compute their abduction of, those of the form $\mathrm{in}(X,Y)$ and those of the form $\mathrm{ta}(X,Y,Z)$ are independent of each other, but those of the form $\mathrm{pa}(X,Y,Z)$ are depended on propositions of the forms $\mathrm{in}(X,Y)$ and $\mathrm{ta}(X,Y,Z)$. So we should compute first the abduction of $\mathrm{in}(X,Y)$ and $\mathrm{ta}(X,Y,Z)$. Now $\mathrm{in}(X,Y)$ is solved by rule (8), $\mathrm{ta}(X,Y,X)$ by rule (2), and as it turned out, when $X \neq Z$, $\mathrm{ta}(X,Y,Z)$ is always false, and its computation is relatively easy. For instance, for the domain with 9 locations, query $\mathrm{ta}(7,1,6)$ took only 2.6 seconds. In comparison, query $\mathrm{pa}(7,1,7)$ took more than 7000 seconds without recycling.

Table 1 contains run time data for some representative queries.[1] 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 $\mathrm{ta}(X,Y,Z)$. As one can see, especially for hard queries like $\mathrm{pa}(7,1,7)$, recycling in this case significantly speeds up the computation.

7 Concluding remarks

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 [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 analysing 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 want to implement a system that can automatically analyse a program and decide how best to recycle previous computations.

[1]Our implementation was written in Sicstus Prolog, and the experiments were done on a PIII 1GHz notebook with 512 MB memory. For generating explanations for regular rewriting system, our implementation is a significant improvement over the one in [Lin and You, 2001]. For instance, for a domain with 7 locations query $\mathrm{pa}(3,2,3)$ 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.

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 1: Recycling in logistics domain. Legends: NR - no recycling; WR - recycling $\mathrm{ta}(X,Y,Z)$ goals. All times are in CPU seconds.

References

[Dershowitz and Jouannaud, 1990] N. Dershowitz and P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Comp. Sci., Vol B: Formal Methods and Semantics*. North-Holland, 1990.

[Dix, 1991] J. Dix. Classifying semantics of logic programs. In *Proc. First Workshop on LPNMR*, pages 167–180, 1991.

[Dung, 1991] P. Dung. Negations as hypothesis: An abductive foundation for logic programming. In Koichi Furukawa, editor, *8th ICLP*. MIT press, 1991.

[Eshghi and Kowalski, 1989] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In *Proc. 6th ICLP*, pages 234–254. MIT Press, 1989.

[Gelfond and Lifschitz, 1988] 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.

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

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

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

[Przymusinski, 1990] T.C. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In *Proc. 7th ICLP*, pages 459–477. MIT Press, 1990.

[Satoh and Iwayama, 1992] K. Satoh and R. Iwayama. A query evaluation method for abductive logic programming. In *Proc. JICSLP'92*, pages 671–685, 1992.

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