ISSN 1369-1961

Department of
Computer Science

Technical Report No. 744

Abductive proofs in dynamic databases

Hisahi Hayashi

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

December 1997

Abductive proofs in dynamic databases

Hisashi Hayashi

Department of Computer Science
Queen Mary and Westfield College, London
Mile End Road, London E1 4NS, U.K.
e-mail: hisashi@dcs.qmw.ac.uk

Abstract

Normal proof procedures in abductive logic programming languages assume that a given program does not change until the proof is completed. However, while a proof is being constructed, new knowledge which affects the proof might be acquired. This paper addresses two important issues: 1. How is it confirmed that the proof being constructed is not affected by the change of the given program? 2. If affected, how is the invalid proof restored? The abductive proof procedure in this paper is based on [EK89, KM90a] and is extended for the preparation of proof checking and proof restoration. Only the propositional case without integrity constraints is considered for simplicity. The predicate case is also discussed informally.

1 Introduction

In most literature on logic programming, the design of a proof procedure does not take into account the dynamic nature of a program. If this kind of proof procedure is used, every time the program is revised and changed, the goal has to be proved again under the new program. However, it is easy to imagine that if the program is changed only a little bit, the old proof is still valid in a lot of cases. The term **dynamic database** is used in this paper to describe a logic program which is subject to change.

Example 1.1 Consider the the following logic program;

p.
q.
$r \leftarrow p$.

Even if the clause 'q.' is removed, the proof for the goal r is not affected. However, if the clause 'p.' is removed, r cannot be proved any more.

Example 1.2 In a lot of countries, there is the principle that

a person is innocent unless proved guilty.

This can be rewritten in a non-monotonic logic such as a logic programming language:

$$innocent(X) \leftarrow not\ guilty(X).$$

Because $guilty(person1)$ cannot be proved, $innocent(person1)$ holds. Even if another sentence:

$$guilty(person2).$$

is added which means that another person, $person2$ is guilty, the proof for $innocent(person1)$ is not affected. However, by adding the information:

$$guilty(person1).$$

this proof becomes invalid because $guilty(person1)$ can now be proved.

The purpose of this project is to reuse the same proof unless it becomes invalid and to change only limited parts of the proof if the proof becomes invalid. Indeed in a lot of cases, it takes time to make a proof. When a whole proof or a part of a proof needs to be constructed within limited time, it is not a good idea to make a brand new proof from the beginning every time programs are corrected.

This paper addresses two important issues in abductive proofs in dynamic databases:

1. How is the validity of a proof checked?
2. How is an invalid proof restored?

Consider planning in robotics. Suppose that a robot is always gathering information through its sensors and that the database of the robot is always changing.

Suppose that the robot starts constructing a plan at time, say, 10:00:00 and finishes at 10:01:15. If the plan is made based only on the information at

10:00:00, it is impossible to say that the plan is still valid at 10:01:15 because the database might be updated between 10:00:00 and 10:01:15. This means that the robot is not sensing the outer world while making the plan. (Even if the robot senses the outer world, it is not using new information during that time.)

Suppose that the robot uses the most recent information available in the database so that it can reflect new information to the plan. For example, when the robot starts constructing a plan at 10:00:00, it uses the database which is valid at 10:00:00 and when the robot completes a half of the plan and tries to make the rest of the plan at 10:00:30, it uses the database which is valid at 10:00:30. Clearly, even in this case, it is not obvious that the plan which is completed at 10:01:15 is valid at 10:01:15 because the plan is still based on the old information (i.e. the database between 10:00:00 and 10:01:15) when it is completed at 10:01:15.

Suppose that the robot completes a correct plan at 10:01:15 and that the robot is executing actions at 10:02:00 based on the plan. Although the plan is correct at 10:01:15, the plan might be useless at 10:02:00. This means that the robot is executing the actions without sensing the outer world believing that the plan is perfect and that unexpected events will never happen.

For this reason, some criteria are needed by which to confirm that the plan is still valid after updating the database. Also it is essential to reuse some parts of the plan to avoid reconstructing a brand new plan and save time.

Let us consider the following more intuitive example. Suppose that some AI researchers are going to an international conference which will be held in Nagoya, Japan from Queen Mary and Westfield College, London. They decided to use the underground, District line and Piccadilly line to go to Heathrow airport. They will take an airplane to Nagoya. From Nagoya airport, they will take an airbus and change to the subway. So far so good. But they haven't completed the whole plan. they are still making a plan for what they should do after arriving at a hotel in Nagoya, which presentations they will listen to, and so on. When still making a plan, they hear the news that the airplane they were planning to use will be cancelled due to a strike. Also they hear the news that the District line is now closed due to signal failures. Obviously, this incomplete plan will not work any more. However, if some parts of this plan are changed, it can still work.

In *partial order planning*, *causal links* or *protected links* are used for replanning. (See standard textbooks such as [RN95].) In this paper, a more general theory which can be applied to any proof in abductive logic

programming will be introduced. Of course it is assumed that the contents of the databases (or programs) being used are updated as the time goes on.

A similar topic is discussed by Shanahan [Sha87a, Sha87b] for the *SLD resolution* case (see standard textbooks such as [Llo84]), where negation cannot be used. The proof procedure discussed in the present paper is for abductive logic programming. The abductive proof procedure adopted in this paper is based on the Eshghi-Kowalski (E-K) procedure [EK89] and the Kakas-Mancarella (K-M) procedure [KM90a]. The E-K procedure is an extension of SLD resolution and the K-M procedure is an extension of the E-K procedure. Only variable free programs and abducibles are investigated at first and later, predicate cases are considered.

The rest of the paper is organised as follows. In Section 2, the E-K and K-M procedures are explained. In Section 3, some applications of abduction are shown. From Section 4 to Section 6, the effects of program updates on proofs are investigated with examples. In Section 7, after a formal extended abductive proof procedure is introduced, a criterion is shown as to when a proof is still valid after the change of the program. From Section 8 to Section 10, the relationships between the proof procedure in Section 7, the K-M procedure, and the E-K procedure are shown and some semantics for the E-K and K-M procedures are discussed. In Section 12, some restoration methods of invalid proofs are investigated. In Section 13, predicates cases are discussed informally.

2 Abductive proof procedure

Various abductive proof procedures have been proposed in the past [EK89, KM90a, DS92, DS, Fun96, FK97, CDT91, SI92, IOH93, Teu93, GMS96]. In this section, an abductive proof procedure is explained briefly.

A literal is either an atom or a sequence of the form: *not* p, where p is an atom. Clauses are of the form $p \leftarrow L_1, ..., L_n.$, where p is an atom and $L_1, ..., L_n$ are literals. Suppose that G is a conjunction of literals (**goals**), Th is a set of clauses (**a program**), and Ab is a set of literals (**abducibles**). The task of an abductive proof procedure in abductive logic programming is to find $\Delta(\subseteq Ab)$ such that Th and Δ are consistent and imply G.

Two kinds of abducibles are used, **positive abducibles** and **negative abducibles**. Positive abducibles are set of atoms undefined in Th. Any negative literal *not* p is regarded as a negative abducible. Negative literals are treated like positive literals. Indeed, in [KM90a] and [EK89], the nega-

tive literal *not* p is replaced by the positive literal p^*. We can assume any positive or negative abducible as a hypothesis.

The proof procedure shown in this section is the abductive proof procedure in [KM90a] which is an extension of [EK89]. The proof procedure in [EK89] is a simulation of SLDNF by abduction and abducibles are restricted to negative abducibles while [KM90a] can use positive abducibles as well as negative abducibles.

The procedure interleaves the **abductive phase**, which is standard SLD-resolution, and the **consistency phase** that incrementally checks that all the hypotheses satisfy

$\forall p(\neg(p \wedge not\ p))$ and
$\forall p(p \vee not\ p)$.

The positive hypotheses Δ_{pos} and the negative hypotheses Δ_{neg} are created incrementally.

This proof procedure will be illustrated by means of examples. Note that abductive phases are computed with sets of literals while consistency phases are computed with sets of sets of literals.

Example 2.1 Consider the following logic program.

clause 1: $p \leftarrow not\ q$.
clause 2: $q \leftarrow not\ r$.

Let us assume r is a positive abducible. A proof for the goal p is as follows;

ab 1 ? p

ab 2 ? *not* q (add *not* q to Δ_{neg})

co 1.1 ? (q)

co 1.2 ? $(not\ r)$

ab 1.2.1 ? r

ab 1.2.2 success (add r to Δ_{pos})

co 1.3 failure

ab 3 success

The goal p succeeds with the positive hypothesis $\Delta_{pos} = \{r\}$ and the negative hypothesis $\Delta_{neg} = \{not\ q\}$ The consistency phase co 1.1 is triggered to check

$$\neg(q \wedge not\ q)$$

and the abductive phase ab 1.2.1 is triggered to satisfy

$$r \vee not\ r.$$

Note that when adding r to Δ_{pos} in the abductive phase ab 1.2.2, the constraint

$$\neg(r \wedge not\ r)$$

is satisfied because $not\ r \notin \Delta_{neg}$.

Example 2.2 Consider the following program.

clause 1: $p \leftarrow not\ q, s.$
clause 2: $q \leftarrow r.$
clause 3: $r \leftarrow s.$

Let us assume s is an abducible. A proof for the goal p is as follows;

ab 1 ? p

ab 2 ? $not\ q, s$

co 1.1 ? (q) (add $not\ q$ to Δ_{neg})

co 1.2 ? (r)

co 1.3 ? (s)

co 1.4 failure (add $not\ s$ to Δ_{neg})

ab 3 ? s

ab 4 failure

The goal p fails at the abductive phase ab 4 because $not\ s \in \Delta_{neg}$ and s cannot be added to Δ_{pos}. If there did not exist $not\ s$ in Δ_{neg}, s could have been added to Δ_{pos} at ab 4 and the goal p could have succeeded, which would have led to an incorrect explanation.

3 Application of abduction

Before discussing about effects of database updates on abductive proofs, some applications of abduction are briefly shown in this section.

In the introduction, *planning* is used as an example of abductive proving. In planning, abduced positive abducibles can be viewed as a plan for a given goal state to be reached in the future [Esh88, Mis91, MDB95].

Abduction is used in other AI areas including: *belief revision* [MKF+84, KM90c, BB95, Bou96]; *database updates* [KM90a, Bry90, CS94]; *temporal explanation* [Sha89, Sha93, DMB92]; *natural language interpretation* [Sti88, HSAM90]; *diagnosis* [Reg83, Rei87, Poo88b, CDT89, PE92]; *default reasoning* [Poo88a].

4 When adding a new clause to the database

If another clause is added to the program, the completed proofs are no longer guaranteed to be valid, because the logic programming languages are non-monotonic. The examples are given already in the introduction. Also if the program is changed or updated when proving some goals, it is meaningless to continue to prove the goals unless it is confirmed that the incomplete proof constructed so far is not affected by the change of the program.

In this section, a criterion is shown as to when the complete and incomplete proof becomes invalid when adding a clause to the program by means of examples.

Example 4.1 Consider Example 2.1. Even if any clause defining the atoms p or r is added to the program, this does not affect the proof because such a clause simply supplies other ways of proving p and r in the abductive phases and p and r are not used in the consistency phase. Note that there never exists a clause to define negative literals in any program.

On the other hand, if a clause which defines the atom q is added to the program, q might be proved using that clause. In this case, the failure in the consistency phase co 1.3 becomes invalid, which leads to the invalidity of the proof.

Consequently, if no clause defining the atom q is added to the program, the proof is still valid.

Example 4.2 Consider Example 2.2. Suppose that the proof is not completed yet but has been constructed up to the consistency phase co 1.2. If, at this moment, the clause

$q \leftarrow t, u.$

is added to the program, it can be easily understood that it has to be proved that either the literal t or the literal u fails.

5 When deleting a clause from the database

In the previous section, the method to detect the invalidity of an abductive proof when adding some clauses to a program was investigated. In this section, the invalidity of the proof when deleting some clauses in the program will be investigated by means of examples.

Example 5.1 Consider Example 2.1. Deleting any clause from the program does not affect the consistency phase because the purpose of the consistency phase is to prove the *failure* and it does not change the *failure* to the *success*.

The proof is affected by deleting a clause from the program if and only if clause 1:

$p \leftarrow not\ q.$

which is used between the two abductive phase ab 1 and ab 2 is deleted.

Note that the abducible r which is added to Δ_{pos} between the two abductive phase ab 1.2.1 and ab 1.2.2 must also be kept in Δ_{pos} afterwards.

Example 5.2 Consider Example 2.2. Suppose that the proof is not yet completed but has been constructed up to the consistency phase co 1.2. If, at this moment, clause 1:

$p \leftarrow not\ q, s.$

is deleted from the program, it can be easily understood that the incomplete proof being constructed so far becomes meaningless. However, deleting any other clause does not affect the incomplete proof.

6 General case

Before generalising the abductive proof procedure which can detect the invalidity of an incomplete or complete proof for the given goals, let us consider slightly more complicated example.

Example 6.1 Consider the following program where d and g are abducibles;

clause 1: $a \leftarrow b, c.$
clause 2: $b \leftarrow d, not\ e.$
clause 3: $c.$
clause 4: $e \leftarrow d, f.$
clause 5: $e \leftarrow g.$
clause 6: $f \leftarrow not\ h.$
clause 7: $f \leftarrow not\ e, g.$
clause 8: $h \leftarrow not\ g, b.$

A proof for the goal a is as follows;

ab 1 ? a

ab 2 ? b, c

ab 3 ? $d, not\ e, c$

ab 4 ? $not\ e, c$ (add d to Δ_{pos})

co 4.1 ? (e) (add $not\ e$ to Δ_{neg})

co 4.2 ? $(d, f), (g)$

co 4.3 ? $(f), (g)$

co 4.4 ? $(not\ e, g), (not\ h), (g)$

co 4.5 ? $(g), (not\ h), (g)$

co 4.6 ? $(not\ h), (g)$ (add $not\ g$ to Δ_{neg})

ab 4.6.1 ? h

ab 4.6.2 ? $not\ g, b$

ab 4.6.3 ? b

ab 4.6.4 ? $d, not\ e$

ab 4.6.5 ? $not\ e$

ab 4.6.6 success

co 4.7 ? (g)

co 4.8 failure

ab 5 ? c

ab 6 success

The goal a succeeds with the hypotheses $\Delta_{pos} = \{d\}$ and $\Delta_{neg} = \{not\ e, not\ g\}$. The abductive phases become invalid if and only if clause 1, clause 2, clause 3, or clause 8 which are used in the abductive phases are deleted from the program or the positive abducible d is deleted from the positive hypotheses Δ_{pos}.

The consistency phases becomes invalid if and only if one or more of the literals e, f, g succeed by the addition of clauses which define them or by the addition of the abducible g to the hypotheses. Note that f is attacked because g is attacked and that e is attacked because f and g are attacked where a literal is **attacked** if and only if its contrary is shown.

Now it can be generalised when complete and incomplete proofs become invalid by database updates.

Observation 6.2 Let *Keep* be the set of all the clauses and the abduced abducibles which are used in the abductive derivations. Let *Reject* be the set of all the literals which need to be proved to be false in the consistency derivations.

The proof is still valid after the change of the program if and only if no clause in *Keep* is deleted and no clause which defines a literal in *Reject* is added.

The formal definitions of abductive derivations and consistency derivations are introduced in the next section.

Example 6.3 Consider Example 6.1. *Keep* and *Reject* can be constructed incrementally as follows.

ab 1 $?a$

ab 2 $?b, c$ (add clause 1 to *Keep*)

ab 3 $?d, not\ e, c$ (add clause 2 to *Keep*)

ab 4 $?not\ e, c$ (add d to *Keep*)

- **co 4.1** $?(e)$ (add $not\ e$ to *Keep*)
- **co 4.2** $?(d, f), (g)$ (add e to *Reject*)
- **co 4.3** $?(f), (g)$
- **co 4.4** $?(not\ e, g), (not\ h), (g)$ (add f to *Reject*)

co 4.5 $?(g), (not\ h), (g)$

co 4.6 $?(not\ h), (g)$ (add g to $Reject$)

ab 4.6.1 $?h$ (add $not\ h$ to $Reject$)

ab 4.6.2 $?not\ g, b$ (add clause 8 to $Keep$)

co 4.6.2.1 $?g$ (add $not\ g$ to $Keep$)

co 4.6.2.2 failure

ab 4.6.3 $?b$

ab 4.6.4 $?d, not\ e$

ab 4.6.5 $?not\ e$

ab 4.6.6 success

co 4.7 $?(g)$

co 4.8 failure

ab 5 $?c$

ab 6 success (add clause 3 to $Keep$)

$Keep$ is

$\{\text{clause 1}, \text{clause 2}, \text{clause 3}, \text{clause 8}, d, not\ e, not\ g\}$.

$Reject$ is

$\{e, f, g, not\ h\}$.

Now $Keep$ plays the role of Δ_{pos} and Δ_{neg}. Although most parts of the proof are the same as the proof in Example 6.1, there is a few differences. From the consistency phase co 4.5 to the consistency phase co 4.6, $not\ g$ is not added to $Keep$. Instead, g is added to $Reject$. Consequently, the consistency phase co 4.6.2.1 is triggered. $Keep$ is constructed only in abductive phases while $Reject$ is constructed only in consistency phases.

Note that any negative literal in $Reject$ is not used in order to detect invalidities of proofs when adding clauses to a program because there does not exist a clause which defines a negative clause.

Observation 6.2 is useful also for incomplete proofs because $Keep$ and $Reject$ can be constructed incrementally and this observation suggests the point to which the proof procedure should backtrack when the given program is updated before completing a proof. The new extended proof procedure

which constructs *Keep* and *Reject* incrementally will be introduced in the next section.

In the predicate case, d has to be added to *Reject* at co 4.3 in the above example. The predicate case will be discussed later. The procedure in the next section will add d to *Keep*.

7 Construction of *Keep* and *Reject*

In this section, an abductive proof procedure which constructs *Keep* and *Reject* is introduced.

Definition 7.1 A **nand set** is a set of literals.

Intuitively, a nand set, $\{p_1, ..., p_n\}$ means $\leftarrow p_1, ..., p_n$..

Definition 7.2 A **goal set** is of the form: (PG, NG, K, R) where PG is a set of literals, NG is a set of nand sets, K is a set of clauses and abducibles, and R is a set of literals.

Note that if $\emptyset \in NG$ then $NG \neq \emptyset$. Intuitively, PG is a set of literals which need to be true. Also for any nand set $\{p_1, ..., p_n\}$ in NG, $\leftarrow p_1, ..., p_n$. needs to be true. (i.e. one of the literals $p_1 ... p_n$ needs to be false.) K and R are used to calculate *Keep* and *Reject*.

Definition 7.3 For a set of literals $S = \{a, L_1, L_2, ..., L_n\}$ and a clause $C = a \leftarrow L_{n+1}, ..., L_m.$, the **resolvent** of S on a by C is the set of literals $\{L_1, ..., L_m\}$.

Definition 7.4 An **abductive derivation** from the goal set G_1 to the goal set G_n under the program P is a sequence of goal sets:

$$\begin{array}{c} (PG_1, \emptyset, K_1, R_1)(= G_1), \\ (PG_2, \emptyset, K_2, R_2)(= G_2), \\ \vdots \\ (PG_n, \emptyset, K_n, R_n)(= G_n) \end{array}$$

such that for any G_i $(1 \leq i \leq n-1)$, there exists a *selected literal* $L_i (\in PG_i)$ such that G_{i+1} is obtained by one of the following **abductive derivation rules**.

a1 L_i is a non-abducible positive literal and there exists a *chosen clause*[1] $C_i(\in P)$ which defines L_i:

PG_{i+1} is the resolvent of PG_i on L_i by C_i. K_{i+1} is obtained by adding C_i to K_i. R_{i+1} is the same as R_i.

a2 L_i is a (positive or negative) abducible in K_i:

PG_{i+1} is $PG_i \setminus \{L_i\}$. K_{i+1} is the same as K_i. R_{i+1} is the same as R_i.

a3 L_i is a positive abducible such that $L_i \notin K_i$, *not* $L_i \notin K_i$, and $L_i \notin R_i$:

PG_{i+1} is $PG_i \setminus \{L_i\}$. K_{i+1} is $K_i \cup \{L_i\}$. R_{i+1} is the same as R_i.

a4 L_i is a negative abducible *not* A_i such that *not* $A_i \notin K_i$, $A_i \notin K_i$, *not* $A_i \notin R_i$, and there exists a consistency derivation from

$$(\emptyset, \{\{A_i\}\}, K_i \cup \{not\ A_i\}, R_i)^2$$

to

$$(\emptyset, \emptyset, K_j, R_j)$$

under P:

PG_{i+1} is $PG_i \setminus \{L_i\}$. K_{i+1} is the same as K_j. R_{i+1} is the same as R_j.

Note that K_i is used to record chosen clauses and abduced abducibles in abductive derivations.

Definition 7.5 A **consistency derivation** from the goal set G_1 to the goal set G_n under the program P is a sequence of goal sets:

$$\begin{array}{c}(\emptyset, NG_1, K_1, R_1)(= G_1),\\ (\emptyset, NG_2, K_2, R_2)(= G_2),\\ \vdots\\ (\emptyset, NG_n, K_n, R_n)(= G_n)\end{array}$$

[1] If the chosen clause is $L_i \leftarrow .$, PG_{i+1} is $PG_i \setminus L_i$.

[2] This goal set is *called* from G_i by a4.

such that for any G_i $(1 \leq i \leq n-1)$, there exist a *selected nand set* $NA_i (\in NG_i)$ and a *selected literal* $L_i (\in NA_i)$, such that G_{i+1} is obtained by one of the following **consistency derivation rules.**

c1 L_i is a non-abducible positive literal which is *not* defined in P:

NG_{i+1} is $NG_i \setminus \{NA_i\}$. K_{i+1} is the same as K_i. R_{i+1} is $R_i \cup \{L_i\}$.

c2 L_i is a non-abducible positive literal which is defined by clauses in P:

Let Cs be all the clauses in P which defines L_i. NG_{i+1} is obtained from NG_i by replacing NA_i with all the nand sets each of which is a resolvent[3] of NA_i on L_i by a clause in Cs. K_{i+1} is the same as K_i. R_{i+1} is $R_i \cup \{L_i\}$.

c3 L_i is a (positive or negative) abducible in K_i:

NG_{i+1} is $NG_i \setminus \{NA_i\} \cup \{NA_i \setminus \{L_i\}\}$. K_{i+1} is the same as K_i. R_{i+1} is $R_i \cup \{L_i\}$.

c4 L_i is a (positive or negative) abducible such that L_i^{*}[4] $\in K_i$:

NG_{i+1} is $NG_i \setminus \{NA_i\}$. K_{i+1} is the same as K_i. R_{i+1} is the same as R_i.

c5 L_i is a positive abducible such that $L_i \notin K_i$ and *not* $L_i \notin K_i$:

NG_{i+1} is $NG_i \setminus \{NA_i\}$. K_{i+1} is the same as K_i. R_{i+1} is $R_i \cup \{L_i\}$.

c6 L_i is a negative abducible *not* A_i such that *not* $A_i \notin K_i$, $A_i \notin K_i$, and there exists an abductive derivation from

$(A_i, \emptyset, K_i, R_i \cup \{not\ A_i\})$[5]

to

[3] If one of the resolvents is $\emptyset$, we cannot derive a complete proof unless we backtrack because $\emptyset$ cannot be resolved.

[4] $(not\ p)^* = p$ and $p^* = not\ p$ where p is an atom

[5] This goal set is *called* from G_i by c6.

$(\emptyset, \emptyset, K_j, R_j)$

under P:

NG_{i+1} is $NG_i \setminus \{NA_i\}$. K_{i+1} is the same as K_j. R_{i+1} is the same as R_j.

Note that R_i is used to record selected literals in consistency derivations.

Definition 7.6 An abductive derivation from $(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$ to $(\emptyset, \emptyset, K_m, R_m)$ under the program P is a **(complete) proof** for the goals $L_1...L_n$ under the program P.

Definition 7.7 An abductive derivation from $(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$ to $(PG_m, \emptyset, K_m, R_m)$ under the program P is an **incomplete proof** for the goals $L_1...L_n$ under the program P.

Note that complete proofs are the limiting case of incomplete proofs.

The following theorem indicates that K is carefully constructed so that it is consistent.

Theorem 7.8 For any incomplete proof under the program P from $(PG_1, \emptyset, \emptyset, \emptyset)$ to (PG_t, NG_t, K_t, R_t), there does not exist an atom p such that $p \in K_t$ and *not* $p \in K_t$.

Proof: See Appendix B.

Observation 7.9 For any complete proof for $L_1...L_n$:

$$(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_m, R_m)$$

under the program P, K_m and R_m are *Keep* and *Reject* of the complete proof respectively.

The next theorem ensures that under certain conditions, it is still possible to use incomplete proofs.

Theorem 7.10 For any incomplete proof for $L_1...L_n$ under the program P:

$$(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_t, \emptyset, K_t, R_t)$$

and for any program Q,

- if all the clauses in K_t are in Q,
- if there does not exist a clause in Q which defines an atom in R_t other than those clauses in P,
- and if there exists an abductive derivation under the program Q:

$$(PG_t, \emptyset, K_t, R_t)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_m, R_m)$$

then there exists a complete proof for $L_1...L_n$ under the program Q:

$$(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(\emptyset, \emptyset, K2_x, R2_x)$$

such that $K2_x \subseteq K_m$ and $R2_x \subseteq R_m$, and $Q \cup \Delta_1 \vdash L_1 \wedge ... \wedge L_n$, where Δ_1 is the set of all the abducibles in K_m.

Proof: See Appendix B.

Note that the above theorem corresponds to Observation 6.2.

Theorem 7.11 For any incomplete proof for $L_1...L_n$ under the program P:

$$(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_t, \emptyset, K_t, R_t)$$

and for any program Q,

- if all the clauses in K_t are in Q,
- if all the clauses in Q which define any atom in R_t are the same as in P,[6]

[6]This condition is stronger than the counterpart of the previous theorem.

- and if there exists an abductive derivation under the program Q:

$$(PG_t, \emptyset, K_t, R_t)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_m, R_m)$$

then the sequence:

$$(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_t, \emptyset, K_t, R_t)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_m, R_m)$$

is a complete proof for $\{L_1, ..., L_n\}$ under the program Q.

Proof: See Appendix B.

Example 7.12 Consider the following program.

clause 1: $a \leftarrow not\ c, b.$
clause 2: $b.$
clause 3: $c \leftarrow d.$
clause 4: $c \leftarrow e.$

A proof for the goal a is shown below.

ab 1 $?a$

ab 2 $?not\ c, b$ (add clause 1 to $Keep$)

co 2.1 $?(c)$ (add $not\ c$ to $Keep$)

co 2.2 $?(d), (e)$ (add c to $Reject$)

co 2.3 $?(e)$ (add d to $Reject$)

co 2.4 failure (add e to $Reject$)

ab 3 $?b$

ab 4 success (add clause 2 to $Keep$)

Even if the clause:

$$a \leftarrow x.$$

is added to the program, the proof is not affected because a does not belong to $Keep$.

If clause 4 is removed, the proof is affected. (See co 2.1 and co 2.2.) However, it is intuitively understandable that this proof is still valid because the removal of clause 4 simply means that $\leftarrow e.$ does not have to be proved, which follows from Theorem 7.10.

Example 7.13 Consider the following program:

clause 1: $a \leftarrow b.$
clause 2: $b \leftarrow c.$
clause 3: $d.$

An incomplete proof for the goal a is shown below.

ab 1 $?a$

ab 2 $?b$ (add clause 1 to $Keep$)

ab 3 $?c$ (add clause 2 to $Keep$)

It is impossible to derive further from the abductive phase ab 3. However, if clause 4: $c \leftarrow d.$ is added to the program, by adding the following abductive derivation to the above incomplete proof, it becomes a complete proof, which follows from Theorem 7.11.

ab 4 $?d$ (add clause 4 to $Keep$)

ab 5 success (add clause 3 to $Keep$)

8 Comparison with the Kakas-Mancarella procedure

In this section, the proof procedure in section 7 is compared with the Kakas-Mancarella procedure [KM90a].

The only difference can be seen in the following example. In the K-M procedure, the absence of the positive literal s in a consistency derivation was recorded by abducing *not* s to avoid abducing s later, while in the procedure

in the previous section, s is recorded in *Reject*. (See the abductive derivation rule a3 and the consistency derivation rule c5.) If c5 is modified so that *not* s is added to *Keep*, the procedure in Section 7 becomes identical to the K-M procedure.

Example 8.1 Consider the following program in Example 2.2.

clause 1: $p \leftarrow not\ q, s.$
clause 2: $q \leftarrow r.$
clause 3: $r \leftarrow s.$

Consider the following proof for p by the K-M procedure, where s is a positive abducible.

ab$_{\mathrm{KM}}$ 1 ? p

ab$_{\mathrm{KM}}$ 2 ? $not\ q, s$

co$_{\mathrm{KM}}$ 1.1 ? (q) (add *not* q to Δ_{neg})
co$_{\mathrm{KM}}$ 1.2 ? (r)
co$_{\mathrm{KM}}$ 1.3 ? (s)
co$_{\mathrm{KM}}$ 1.4 failure (add *not* s to Δ_{neg})

ab$_{\mathrm{KM}}$ 3 ? s

ab$_{\mathrm{KM}}$ 4 failure

"failure" was derived from ab 3 to ab 4 because $not\ s \in \Delta_{neg}$.

The corresponding proof by the procedure in Section 7 is as follows, where s is a positive abducible.

ab 1 ? p

ab 2 ? $not\ q, s$ (add clause 1 to *Keep*)

co 1.1 ? (q) (add *not* q to *Keep*)
co 1.2 ? (r) (add q to *Reject*)
co 1.3 ? (s) (add r to *Reject*)
co 1.4 failure (add s to *Reject*)

ab 3 ? s

ab 4 failure

The goal p fails at the abductive phase ab 4 because $s \in Reject$ and s cannot be added to $Keep$ at ab 4. If s was not in $Reject$, s could have been added to $Keep$ at ab 4 and the goal p could have succeeded, which would have led to an incorrect explanation.

If *not* s is regarded as an explicit negation of s, recording the absence of the positive abducible s by abducing *not* s is too strong as Kakas, Kowalski and Toni discuss in [KKT93, KKT97].

A similar comparison can be made in Example 6.1 and Example 6.3, as is discussed in Example 6.3.

9 Comparison with the Eshghi-Kowalski procedure

In this section, the proof procedure in section 7 is compared with the Eshghi-Kowalski procedure [EK89] which the K-M procedure is based on.

Let P be a program. After successfully proving the goals G with the set of positive abducibles Δ_{pos} and the set of negative abducibles Δ_{neg} ($\Delta_{pos} \cup \Delta_{neg} \subseteq Keep$) by the procedure in Section 7, it is possible to reprove the goal G by the E-K procedure with Δ_{neg}, by treating $P \cup \Delta_{pos}$ as the given program. The reason is as follows.

In E-K procedures, there does not exist any abductive or consistency derivation rule for positive abducibles. It suffices to show that there exists an alternative derivation rule available which has the same effect. In the E-K procedure, the abductive derivation rules a2 and a3 are not applied to any positive literal p in Δ_{pos}. However, a1 can be applied instead and the effect is the same. Similarly the consistency derivation rules c3 and c5 are not applied to p in the E-K procedure. However, c2 and c1 can be applied respectively instead and the effect is the same. The consistency derivation rule c4 is not applied to any positive literal q in the E-K procedure. However, c1 can be applied to q instead because q was not abduced by the procedure in Section 7 if c4 was applied to the positive abducible q. (See Theorem 7.8.) The effect is also the same.

Consequently, $(P \cup \Delta_{pos}) \cup \Delta_{neg} \vdash G$ by the E-K procedure.

Example 9.1 Consider the following program P where b and c are positive abducibles.

clause1: $a \leftarrow not\ b, c.$

The proof procedure in Section 7 successfully proves the goal a under the program P as follows. The set of abduced abducibles $\{not\ b, c\}$ is a subset of $Keep$.

ab 1 $?a$

ab 2 $?not\ b, c$ (add clause 1 to $Keep$)

co 2.1 $?(b)$ (add $not\ b$ to $Keep$)

co 2.2 failure (add b to $Reject$)

ab 3 $?c$

ab 4 success (add c to $Keep$)

The E-K procedure successfully proves the goal a with $\{not\ b\}$ under the program $P \cup \{c.\}$ as follows.

ab$_{\mathbf{EK}}$ 1 $?a$

ab$_{\mathbf{EK}}$ 2 $?not\ b, c$

co$_{\mathbf{EK}}$ 2.1 $?(b)$ (add $not\ b$ to Δ)

co$_{\mathbf{EK}}$ 2.2 failure

ab$_{\mathbf{EK}}$ 3 $?c$

ab$_{\mathbf{EK}}$ 4 success

Note that now $c.$ is a clause in the program and b is not an abducible.

10 Semantics for the Eshghi-Kowalski procedure

In this section, the semantics for the E-K procedure is discussed.

In the original paper describing the E-K procedure [EK89], a declarative semantics which has one-to-one correspondence to *stable model semantics* [GL88] is defined. However, this semantics is not always correct. Indeed, there exist programs which do not have any stable models. Several other semantics which conquer this problem were defined as follows.

One approach is to adopt an argumentation-theoretic interpretation. Dung [Dun91] defined *preferred extension semantics* and showed that it is

sound. Later Kakas and Mancarella [KM93] showed that it is equivalent to *partial stable model semantics* [SZ90].

An interesting three-valued semantics is *finite failure stable model semantics* [GMS93]. It is defined in the style of stable model semantics and it builds on an underlying *Kripke/Kleene semantics* [Fit85, Kun87a]. This semantics can capture finite failure while *extended stable model semantics* [Prz90] can capture infinite failure.

11 Semantics for the Kakas-Mancarella procedure

As is mentioned earlier, the K-M procedure is an extension of the E-K procedure. If no positive abducible is used, the K-M procedure is the same as the E-K procedure.

In the original paper [KM90a], Kakas and Mancarella used *generalised stable model semantics* [KM90b]. In this semantics, they adopted stable model semantics for a given program and abduced abducibles. Note that it is possible to generalise other semantics in this way. The problem is that there are cases where there does not exist any stable model.

Toni [Ton95] reinterpreted the K-M procedure in argumentation-theoretic term as in the E-K procedure case. This semantics is similar to the notion of admissibility for extended logic programming [Dun93] as is pointed out in [KKT97].

Three-valued completion semantics can be used for the K-M procedure as is adopted as a semantics [DS93] for the latest version [DS] of SLD-NFA [DS92]. This semantics is based on three-valued completion semantics for non-abductive logic programs [Kun87b] and two-valued completion semantics for abductive logic programs [CDT91]. Fung and Kowalski [Fun96, FK97] also use three-valued completion semantics for their iff proof procedure for abductive logic programming.

Any above semantics can be adopted. The investigation of semantics is not the main theme of this project.

12 Restoration of proofs

From Theorem 7.10, it is possible to judge if complete and incomplete proofs are still valid under modified programs. However, it is not discussed yet how the validity to a proof is restored using parts of the invalid old proof. In this section, some restoration methods are introduced.

The first restoration method is the simplest backtracking method.

Definition 12.1 Let S: $(PG_1, NG_1, K_1, R_1)...\ (PG_i, NG_i, K_i, R_i)$ be an incomplete proof under the program P. Let Q be a program.

A **deleted keeping clause** in S when changing from P to Q is a clause in K_i which does not exist in Q.

An **added rejecting clause** r in S when changing from P to Q is a clause which defines a positive literal in R_i such that r exists in Q but not in P.

A **deleted rejecting clause** r in S when changing from P to Q is a clause which defines a positive literal in R_i such that r exists in P but not in Q.

Definition 12.2 Let $S : G_1...G_x...G_n$ be an incomplete proof under the program P. Let Q be a program. The goal set $(PG_x, NG_x, K_x, R_x)(= G_x)$ is the **backtrack point** for S when changing from P to Q, if x is the least y such that one of the following holds:

- The chosen clause for G_y is a deleted keeping clause in S when changing from P to Q.
- There exists an atom in R_y which is defined by an added rejecting clause in S when changing from P to Q.
- There exists an atom in R_y which is defined by a deleted rejecting clause in S when changing from P to Q.[7]

Definition 12.3 Let S:

$$\begin{array}{c} (PG_1, \emptyset, \emptyset, \emptyset) \\ \vdots \\ (PG_x, NG_x, K_x, R_x) \\ \vdots \\ (PG_i, NG_i, K_i, R_i) \end{array}$$

be an incomplete proof under the program P. Let Q be a program. A **backtrack resumption** of S when changing from P to Q is any sequence of the form:

[7]This condition is not really needed. However, if we add this condition, we can understand the following backtracking method easier.

$$(PG_1, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_x, NG_x, K_x, R_x)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_j, R_j)$$

where (PG_x, NG_x, K_x, R_x) is the backtrack point for S when changing from P to Q and the sequence:

$$(PG_x, NG_x, K_x, R_x)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_j, R_j)$$

is an abductive derivation under Q.

Theorem 12.4 Any backtrack resumption of an incomplete proof for the goals $L_1 ... L_h$ under the program P when changing from P to the program Q is a complete proof for the goals $L_1 ... L_h$ under Q.

Proof: See Appendix B.

Example 12.5 Consider the following program in Example 7.12.

clause 1: $a \leftarrow not\ c, b.$
clause 2: $b.$
clause 3: $c \leftarrow d.$
clause 4: $c \leftarrow e.$

and consider the following proof for the goal a shown in Example 7.12.

ab 1 $?a$

ab 2 $?not\ c, b$ (add clause 1 to $Keep$)

co 2.1 $?(c)$ (add $not\ c$ to $Keep$)

co 2.2 $?(d), (e)$ (add c to $Reject$)

co 2.3 $?(e)$ (add d to $Reject$)

co 2.4 failure (add e to $Reject$)

ab 3 $?b$

ab 4 success (add clause 2 to $Keep$)

If the clause:

$e.$

is added to the program and if clause 2:

$b.$

is removed from the program, both the consistency derivation from co 2.3 to co 2.4 and the abductive derivation from ab 3 to ab 4 are affected. The backtrack point[8] is ab 2.

The method described above is simple. However, it is necessary to record the whole derivation in order to find the backtrack point. Also it abandons all parts of the old proof after the backtrack point. The next method overcomes those problems.

Theorem 12.6 Let S :

$$(\{L_1 ... L_u\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_i, \emptyset, K_i, R_i)$$

be an incomplete proof under the program P. Let Q be a program. Let DKC be the set of all the deleted keeping clauses in S when changing from P to Q. Let $\{c_1, ..., c_n\}$ be the set of all the literals which are defined by DKC. Let $\{a_1, ..., a_m\}$ be the set of all the literals which are defined by added rejecting clauses in S when changing from P to Q.

If there exists an abductive derivation:

$$(PG_i \cup \{c_1, ..., c_n\}, \emptyset, K_i \setminus DKC, R_i)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_j, R_j)$$

under Q and for such K_j and R_j, there exists a consistency derivation:

$$(\emptyset, \{\{a_1\}, ..., \{a_m\}\}, K_j, R_j)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_k, R_k)$$

[8]It is possible to change the definition of backtrack points so that co 2.3 becomes the backtrack point.

under Q, then there exists a complete proof:

$$(\{L_1 ... L_u\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(\emptyset, \emptyset, K2_x, R2_x)$$

under Q, such that $K2_x \subseteq K_k$ and $R2_x \subseteq R_k$, and $Q \cup \Delta_1 \vdash L_1 \wedge ... \wedge L_u$, where Δ_1 is the set of all the abducibles in K_k.

Proof: See Appendix B.

Example 12.7 Consider Example 7.12. (The program and the proof are recapped in Example 12.5.) If the clauses:

$e \leftarrow f.$
$b \leftarrow not\ d.$

are added to the program and if clause 2:

$b.$

is removed from the program, both the consistency derivation from co 2.3 to co 2.4 and the abductive derivation from ab 3 to ab 4 are affected.

If there exists an abductive derivation from $(\{b\}, \emptyset, \{\text{clause1}, not\ c\}, \{c, d, e\})$ to $(\emptyset, \emptyset, K_i, R_i)$ and if there exists a consistency derivation from $(\emptyset, \{\{e\}\}, K_i, R_i)$ to $(\emptyset, \emptyset, K_j, R_j)$, then there exists a complete proof from $(\{a\}, \emptyset, \emptyset, \emptyset)$ to $(\emptyset, \emptyset, K2_x, R2_x)$ such that $K2_x \subseteq K_j$ and $R2_x \subseteq R_j$.

The method shown in Theorem 12.6 does not require recording the whole derivation and most parts of the old proof are used. The aim of this strategy is to supplement the old proof without backtracking

However, because it is not possible to track the relationships between literals, it is unavoidable to prove meaningless subgoals which are derived by deleted keeping clauses.

This problem arises when deleting clauses from the program.

Example 12.8 Consider the following program.

clause 1: $a \leftarrow b.$
clause 2: $a \leftarrow not\ x.$
clause 3: $b \leftarrow c.$
clause 4: $c \leftarrow d.$
clause 5: $d.$

A proof for the goal a is constructed as follows.

ab 1 $?a$

ab 2 $?b$ (add clause 1 to *Keep*)

ab 3 $?c$ (add clause 3 to *Keep*)

Suppose that when this incomplete proof is constructed, the program is corrected and clause 1 is deleted. Using Theorem 12.6, it is possible to restore the proof by reproving a and c. However, it is meaningless to reprove c because c does not imply a any more.

By restricting the program updates only to the addition of clauses, the following corollary holds.

Corollary 12.9 Suppose that P and Q are programs such that $P \subseteq Q$. Let S :

$$(\{L_1 ... L_u\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(PG_i, \emptyset, K_i, R_i)$$

be an incomplete proof for the goals under P. Let $\{a_1, ..., a_m\}$ be the set of all the literals which are defined by the added rejecting clauses in S when changing from P to Q.

If there exists a consistency derivation:

$$(\emptyset, \{\{\ a_1\}, ..., \{a_m\}\}, K_i, R_i)$$
$$\vdots$$
$$(\emptyset, \emptyset, K_k, R_k)$$

under Q, then there exists a complete proof:

$$(\{L_1 ... L_u\}, \emptyset, \emptyset, \emptyset)$$
$$\vdots$$
$$(\emptyset, \emptyset, K2_x, R2_x)$$

under Q, such that $K2_x \subseteq K_k$ and $R2_x \subseteq R_k$, and $Q \cup \Delta_1 \vdash L_1 \wedge ... \wedge L_u$, where Δ_1 is the set of all the abducibles in K_k.

Proof: See Appendix B.

The restoration method in Corollary 12.9 does not trigger any redundant computation like in Theorem 12.6. As a matter of course, it is not necessary to record the whole derivation like Theorem 12.4. Note that no clause except abduced literals has to be recorded in $Keep$ if no clause is deleted from the given program. The deletion of clauses can be simulated by the addition as is shown in Appendix A.

Furthermore, Theorem 12.6 and Theorem 12.4 can be combined. In this strategy, even if the atom p cannot be reproved directly in an abductive derivation, it is still possible to reprove a selected literal from which p is derived. Similarly, even if $\leftarrow L$ cannot be reproved directly in a consistency derivation, it is still possible to reprove $\leftarrow p$ instead where p is a selected literal from which L is derived.

This intuitive method is explained by the following example.

Example 12.10 Consider the following program.

clause 1: $a \leftarrow b$.
clause 2: $b \leftarrow c, d$.
clause 3: $c \leftarrow e$.
clause 4: $c \leftarrow h$.
clause 5: $d \leftarrow not\ g$.
clause 6: $e \leftarrow f$.
clause 7: f.
clause 8: $g \leftarrow k$.
clause 9: h.
clause 10: $k \leftarrow i, j$.

A proof for the goal a is as follows;

ab 1 ? a

ab 2 ? b (add clause 1 to $Keep$)

ab 3 ? c, d (add clause 2 to $Keep$)

ab 4 ? e, d (add clause 3 to $Keep$)

ab 5 ? f, d (add clause 6 to $Keep$)

ab 6 ? d (add clause 7 to $Keep$)

ab 7 ? $not\ g$ (add clause 5 to $Keep$)

co 7.1 ? (g) (add *not* g to $Keep$)

co 7.2 ? (k) (add g to $Reject$)

co 7.3 ? (i, j) (add k to $Reject$)

co 7.4 failure (add i to $Reject$)

ab 8 success

The proof succeeds with:

- $Keep = \{$ clause 1, clause 2, clause 3, clause 5, clause 6, clause 7, *not* $g\}$
- $Reject = \{g, k, i\}$

Suppose that the clause: $i.$ is added to the program and that clause 7: $f.$ is removed from the program.

There is no abductive derivation from $(\{f\}, \emptyset, Keep, Reject)$ to $(\emptyset, \emptyset, K_1, R_1)$, nor from $(\{e\}, \emptyset, Keep, Reject)$ to $(\emptyset, \emptyset, K_1, R_1)$, however, there is an abductive derivation from $(\{c\}, \emptyset, Keep, Reject)$ to $(\emptyset, \emptyset, K_1, R_1)$. Also there is no consistent derivation from $(\emptyset, \{\{i\}\}, K_1, R_1)$ to $(\emptyset, \emptyset, K_2, R_2)$, however, there is a consistency derivation from $(\emptyset, \{\{k\}\}, K_1, R_1)$ to $(\emptyset, \emptyset, K_2, R_2)$.

Consequently, there is a complete proof for a under the new program.

In order to use the above method, *trees* which express relationships of resolutions between literals rather than sequences of goal sets are required. In the above example, f at ab 5 is derived from e at ab 4 which is derived from c at ab 3. i at co 7.3 is derived from k at co 7.2. Furthermore, a literal in a goal set G_i in a derivation is not always derived from a literal in the previous goal set G_{i-1}. For example, d at ab 6 is derived from b at ab 2 but not from a literal at ab 5.

Obviously, recording and searching trees take time and memories. The comparison of proof restoration methods in terms of efficiency needs further investigation.

13 The predicate case

In this section, the methods to find and restore the invalidities of proofs in the predicate case will be shown intuitively. It is assumed that all variables in negative literals are substituted with ground terms before they are resolved so that they do not flounder.

Although the same strategy as the propositional case can be applied to the predicate case, it is necessary to extend the definition of $Keep$ and $Reject$ slightly.

In the propositional case, $Keep$ is the set of abduced abducibles and clauses used for resolutions. There are two roles of $Keep$: 1: to record abduced abducibles; 2: to check if the clauses used for resolutions in abductive derivations are not deleted from the program. In the predicate case, the role of $Keep$ is almost the same. However, recording only clauses used in abductive derivations is not enough because in order to apply those clauses, the heads of those clauses are unified with the positive literals to be resolved. Those positive literals have to be recorded to remember what literals should be proved in abductive derivations.

Example 13.1 Consider the following program.

clause 1: $mortal(X) \leftarrow animal(X)$.
clause 2: $animal(X) \leftarrow human(X)$.
clause 3: $human(X) \leftarrow philosopher(X)$.
clause 4: $philosopher(s)$.
clause 5: $human(t)$.
clause 6: $human(s)$.

A proof for the goal $mortal(s)$ is shown below.

ab 1 $?mortal(s)$

ab 2 $?animal(s)$ (add ($mortal(s)$, clause 1) to $Keep$)

ab 3 $?human(s)$ (add ($animal(s)$, clause 2) to $Keep$)

ab 4 $?philosopher(s)$ (add ($human(s)$, clause 3) to $Keep$)

ab 5 success (add ($philosopher(s)$, clause 4) to $Keep$)

Unless a clause in $Keep$ is removed, this proof is valid. (cf. Theorem 7.10) If clause 3 is removed, this proof becomes invalid and this invalidity is detected because clause 3 is recorded in $Keep$. However, without recording the literal $human(s)$ which was resolved using clause 3, $human(s)$ cannot be reproved using clause 6 like the propositional case because in clause 3, $human(X)$ is not ground and the term with which X was unified is unknown. (cf. Theorems 12.6) Obviously, it is meaningless to prove $human(t)$ using clause 5. By recording the selected literal $human(s)$ as well as the chosen clause, clause 3, $human(s)$ can be proved.

As shown above, it is not difficult to reprove a selected literal which was proved by a deleted keeping clause. More complicated problem arises when re-substituting variables.

Example 13.2 Consider the following program.

clause 1: $a(X) \leftarrow b(X)$.
clause 2: $b(X) \leftarrow c(X), d(X)$.
clause 3: $c(s)$.
clause 4: $c(t)$.
clause 5: $d(s)$.

A proof for the goal $a(X)$ is shown below.

ab 1 $?a(X)$

ab 2 $?b(X)$ (add ($a(X)$, clause 1) to $Keep$)

ab 3 $?c(X), d(X)$ (add ($b(X)$, clause 2) to $Keep$)

ab 4 $?d(s)$ (add ($c(X)$, clause 3) to $Keep$)

ab 5 success (add ($d(s)$, clause 5) to $Keep$)

$?a(X)$ succeeds with the substitution $X = s$. When clause 3 is deleted, this proof becomes invalid. Because clause 3 is recorded in $Keep$, this invalidity is found and $c(X)$ can be successfully reproved with the substitution $X = t$. However, $b(t)$ and $a(t)$ are not proved yet because $d(t)$ is not proved.

This substitution problem is more difficult. Indeed, re-substitution of X requires reproving $d(X)$ in the above example. This suggests to record not only the selected literals but also the whole subgoals.

Example 13.3 Consider the proof in Example 13.2. $Keep$ can be constructed as follows so that it can include subgoals.

ab 2 add ($[a(X)]$, clause 1) to $Keep$

ab 3 add ($[b(X)]$, clause 2) to $Keep$

ab 4 add ($[c(X), d(X)]$, clause 3) to $Keep$

ab 5 add ($[d(s)]$, clause 5) to $Keep$

The idea is that every time a chosen clause is erased, all the subgoals are reproved which are in the same tuple as the deleted chosen clause is an element of. The problem is that it takes more memories.

Construction of *Reject* in the predicate case is similar to the propositional case. First of all, as in the propositional case, it is necessary to record the selected literals which are (being) proved not to be true.

Example 13.4 Consider the following program.

clause 1: $innocent(X) \leftarrow not\ guilty(X)$.
clause 2: $guilty(X) \leftarrow law(L), against(X, L)$.
clause 3: $law(eu)$.
clause 4: $law(uk)$.
clause 5: $against(a, japan)$.

A proof for the goal $innocent(b)$ is shown below.

ab 1 $?innocent(b)$

ab 2 $?not\ guilty(b)$

co 2.1 $?(guilty(b))$ (add $not\ guilty(b)$ to $Keep$)

co 2.2 $?(law(L), against(b, L))$ (add $guilty(b)$ to $Reject$)

co 2.3 failure (add $against(b, L)$ to $Reject$)

ab 3 success

The only elements of *Reject* are $guilty(b)$ and $against(b, L)$. Unless clauses whose heads can unify with these two atoms are added, this proof is still valid. Even if the clause:

$against(c, uk)$.

is added to the program, this proof is still valid because it does not unify with $against(b, L)$. If the clause:

$guilty(X) \leftarrow special\_case(X)$.

which unifies with $guilty(b)$ is added to the program, $\leftarrow special\_case(b)$. has to be proved.

One of the big differences between the propositional case and the predicate case is the fact that proving in the predicate case largely depends on unification. The definition of *Reject* needs to be corrected.

Example 13.5 Consider the program in Example 13.4. A proof for the goal $innocent(a)$ is shown below.

ab 1 $?innocent(a)$

ab 2 $?not\ guilty(a)$

co 2.1 $?(guilty(a))$ (add $not\ guilty(a)$ to $Keep$)

co 2.2 $?(law(L), against(a, L))$ (add $guilty(a)$ to $Reject$)

co 2.3 $?(against(a, eu)), (against(a, uk))$ (add $law(L)$ to $Reject$)

co 2.4 $?(against(a, uk))$ (add $against(a, eu)$ to $Reject$)

co 2.5 failure (add $against(a, uk)$ to $Reject$)

ab 3 success

If $law(un)$ is added to the program, this proof becomes invalid and this invalidity is detected because $law(L)$ is in $Reject$. However, what needs to be proved is not $\leftarrow law(L)$. but $\leftarrow law(L), against(a, L)$..

The above example suggests to record in $Keep$ not only the selected literals in consistency derivations but also the selected nand sets like Example 13.3. This strategy is shown below.

Example 13.6 Consider the proof in Example 13.5. $Reject$ can be constructed as follows so that it includes selected nand sets.

co 2.2 add $(guilty(a), [guilty(a)])$ to $Reject$

co 2.3 add $(law(L), [law(L), against(a, L)])$ to $Reject$

co 2.4 add $(against(a, eu), [against(a, eu)])$ to $Reject$

co 2.5 add $(against(a, uk), [against(a, uk)])$ to $Reject$

The idea is whenever any clause which defines a selected literal in $Reject$ is added, one of the literals in the selected nand set which the selected literal belongs to is reproved. The problem is that it takes memories.

Now it is understood that the counterparts of Theorem 12.6 and Corollary 12.9 in the predicate case can be made.

As in the propositional case, further investigation is needed for restoration methods in the predicate case. The memory space used, the efficiency, and the number of invalid proofs which can be restored need to be taken into account.

14 Conclusion

In Section 7, a criterion was shown as to whether the continuation of the proof procedure will result in a correct proof or not. Even if it is not clear if the proof being constructed is valid or not, it is still possible to restore the proof by the methods shown in Section 12. In Section 13, it was shown intuitively how this theory can be extended to the predicate case.

Although some proof restoration methods were introduced in this paper, more methods can be investigated in both the propositional cases and the predicate cases.

15 Future work

There is a lot of future work for this project. First, the formal extension to the predicate case will be investigated. Second, more proof restoration methods will be presented and compared. When restoration methods are compared, the efficiency, the memory space to be used, and the number of invalid proofs which can be restored will be checked.

After that, two avenues of further work will be considered. One topic is to investigate constructive negations and constraint logic programming. Constructive negations are used in some abductive proof procedures [DS92, DS, Fun96, FK97, Teu93]. Abductive constraint logic programming is investigated in [Bür91, Bür94, Mai92, KM95, KM97]. Constructive negations are important if uninstantiated negative literals are selected. For example, in planning, if future time points are expressed by means of variables, constructive negations are needed because negative literals containing these variables are used and these are not always instantiated in order to construct *partial order plans*. Constraint logic programming is also important for planning because in partial order planning, the constraint $T_1 < T_2$ needs to be calculated, where T_1 and T_2 are variables expressing the future time points. In order to conquer these problems, some abductive procedures [Sha89, Esh88] specialise in temporal reasoning while Dryllerakis [Dry92] uses the constraint logic programming CLP($\mathcal{R}$).

Another topic is to apply the theory presented in this paper to such areas as robotics and planning. As mentioned in the introduction, this project is closely related to planning in robotics. However, in order to apply the theory to such applications, the theoretical work required for the applications should be investigated. For example, constraint logic programming has to

be investigated if calculations such as $1+1 < 3$ are needed during abduction.

Acknowledgement

I am grateful to Dr. Murray Shanahan for draft checking.

References

[BB95] C. Boutilier and V. Becher. Abduction as belief revision. *Artificial intelligence*, 77:43–94, 1995.

[Bou96] C. Boutilier. Abduction to plausible causes: an event-based model of belief update. *Artificial intelligence*, 83:143–166, 1996.

[Bry90] F. Bry. Intensional updates: abduction via deduction. In *International conference on logic programming*, pages 561–575, 1990.

[Bür91] J. H. Bürckert. On abduction and answer generation through constraint resolution. *Technical report-deliverable of ESPRIT project COMPULOG I*, 1991.

[Bür94] J. H. Bürckert. A resolution principle for constrained logics. *Artificial intelligence*, 66:235–271, 1994.

[CDT89] L. Console, D. T. Duprè, and P. Torasso. A theory for diagnosis for incomplete causal models. In *International joint conference on artificial intelligence*, pages 1311–1317, 1989.

[CDT91] L. Console, D. T. Duprè, and P. Torasso. On the relationship between abduction and deduction. *Journal of logic and computation*, 1(5):661–690, 1991.

[CS94] L. Console and M. L. Sapino. The role of abduction in database view updating. *Journal of intelligent information systems*, 1994.

[DMB92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *European conference on artificial intelligence*, pages 384–388, 1992.

[Dry92] Dryllerakis. Results from implementing the event calculus in CLP($\mathcal{R}$). Technical report, Department of Computing, Imperial College, London, 1992.

[DS] M. Denecker and D. D. Schreye. SLDNFA: an abductive procedure for normal abductive programs. Department of computer science, K. U. Leuven.

[DS92] M. Denecker and D. D. Schreye. SLDNFA: an abductive procedure for normal abductive programs. In *the joint international conference and symposium on logic programming*, pages 686–700, 1992.

[DS93] M. Denecker and D. D. Schreye. Justification semantics: unifying framework for the semantics of logic programs. In *Logic programming and non-monotonic reasoning workshop*, pages 365–379, 1993.

[Dun91] P.M. Dung. Negation as hypothesis: an abductive foundation for logic programming. In *International conference on logic programming*, 1991.

[Dun93] P. M. Dung. An argumentation semantics for logic programming with explicit negation. In *International conference on logic programming*, 1993.

[EK89] E. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *International conference on logic programming*, 1989.

[Esh88] K. Eshghi. Abductive planning with event calculus. In *International conference and symposium on logic programming*, pages 562–579, 1988.

[Fit85] M. Fitting. A Kripke/Kleene semantics for logic programs. *Journal of logic programming*, 2:295–312, 1985.

[FK97] T. H. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *Journal of logic programming*, 1997.

[Fun96] T. H. Fung. *Abduction by deduction.* PhD thesis, Imperial College, London, 1996.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International conference and symposium on logic programming*, pages 1070–1080, 1988.

[GMS93] L. Giordano, A. Martelli, and M. L. Sapino. A semantics for Eshghi and Kowalski's abductive procedure. In *International conference on logic programming*, pages 586–600, 1993.

[GMS96] L. Giordano, A. Martelli, and M. L Sapino. Extending negation as failure by abduction: a three-valued stable model semantics. *Journal of logic programming*, 26:31–67, 1996.

[HSAM90] J. R. Hobbs, M. Stickel, D. Appelt, and P. Martin. Interpretation as abduction. Technical Report 499, SRI, 1990.

[IOH93] K. Inoue, Y. Ohta, and R. Hasegawa. Bottom-up abduction by model generation. Technical Report TR-816, Institute for new generation computer technology, 1993.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of logic and computation*, 2(6):719–770, 1993.

[KKT97] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of logic in artificial intelligence and logic programming*, 5, 1997.

[KM90a] A. C. Kakas and P. Mancarella. Database updates through abduction. In *VLDB Conference*, pages 650–661, 1990.

[KM90b] A. C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *European conference on artificial intelligence*, pages 385–391, 1990.

[KM90c] A. C. Kakas and P. Mancarella. Knowledge assimilation and abduction. In *European conference on artificial intelligence, international workshop on truth maintenance*, 1990.

[KM93] A. C. Kakas and P. Mankarella. Preferred extensions are partial stable models. *Journal of logic programming*, 14(3, 4):341–348, 1993.

[KM95] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *International conference on logic programming*, pages 399–413, 1995.

[KM97] A. C. Kakas and C. Mourlas. ACLP: Flexible solutions to complex problems. In *International conference on logic programming and non-monotonic logic*, pages 387–398, 1997.

[Kun87a] K. Kunen. Negation in logic programming. *Journal of logic programming*, 4:289–308, 1987.

[Kun87b] K. Kunen. Negation in logic programming. *journal of logic programming*, 4(3):289–308, 1987.

[Llo84] J. W. Lloyd. *Foundations of logic programming.* Springer Verlag, 1984.

[Mai92] E. Maim. Abduction and constraint logic programming. In *European conference on artificial intelligence*, pages 149–153, 1992.

[MDB95] L. R. Missiaen, M. Denecker, and M. Bruynooghe. CHICA, an abductive planning system based on event calculus. *Journal of logic and computation*, 5(5):579–602, 1995.

[Mis91] L. R. Missiaen. Localized abductive planning for robot assembly. In *IEEE conference on robotics and automation*, pages 605–610, 1991.

[MKF+84] T. Miyaki, S. Kitakami, H. Furukawa, A. Takeuchi, and H. Yokota. A knowlege assimilation method for logic databases. In *International symposium on logic programming*, pages 118–125, 1984.

[PE92] C. Preist and K. Eshghi. Consistency-based and abductive diagnoses as generalised stable models. In *International conference on fifth generation computer systems*, pages 514–521, 1992.

[Poo88a] D. Poole. A logical framework for default reasoning. *Artificial intelligence*, 36:27–47, 1988.

[Poo88b] D. Poole. Representing knowledge for logic-based diagnosis. In *International conference on fifth generation computer systems*, pages 1282–1290, 1988.

[Prz90] T. C. Przymusinska. Extended stable semantics for normal and disjunctive programs. In *International conference on logic programming*, pages 459–477, 1990.

[Reg83] J. Reggia. Diagnostic expert systems based on a set-covering model. *International journal of man-machine studies*, 19(5):437–460, 1983.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32:57–95, 1987.

[RN95] S. Russell and P. Norvig. *Artificial intelligence: a modern approach.* Prentice Hall, 1995.

[Sha87a] M. Shanahan. *Exploiting dependencies in search and inference mechanisms.* PhD thesis, King's College, University of Cambridge, 1987.

[Sha87b] M. Shanahan. An incremental theorem prover. In *International joint conference on artificial intelligence*, pages 987–989, 1987.

[Sha89] M. Shanahan. Prediction is deduction but explanation is abduction. In *International joint conference on artificial intelligence*, pages 1055–1060, 1989.

[Sha93] M. Shanahan. Explanation in the situation calculus. In *International joint conference on artificial intelligence*, pages 160–165, 1993.

[SI92] K. Satoh and N. Iwayama. A query evaluation method for abductive logic programming. In *International joint conference and symposium on logic programming*, pages 671–685, 1992.

[Sti88] M. E. Stickel. A prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. In *International computer science conference (artificial intelligence: theory and applications)*, pages 343–350, 1988.

[SZ90] D. Saccà and C. Zaniolo. Stable models and non determinism for logic programs with negation. In *ACM SIGMOD-SIGACT symposium on principles of database systems*, pages 205–217, 1990.

[Teu93] F. Teusink. Using SLDFA-resolution with abductive logic programs. In *ILPS93 post-conference workshop on logic programming with incomplete information*, pages 35–47, 1993.

[Ton95] F. Toni. A semantics for Kakas-Mancarella procedure for abductive logic programming. In *GULP95*, pages 231–242, 1995.

Appendix A

A new method to simulate deletion of a clause by the addition of an atom is introduced in this section. The motivation is to use Corollary 12.9.

Definition 15.1 Given a set of clauses P, $P!$ is the least set of clauses such that for any clause in P:

$$a \leftarrow L_1, ..., L_n.$$

$P!$ contains the clause:

$$a \leftarrow L_1, ..., L_n, not\ a_{del}.$$

where a_{del} is a new atom called a **deletion atom**.

Note that any deletion atom in a clause in the program $P!$ must not occur in any other clause in $P!$.

Definition 15.2 Whenever a clause in P:

$$a \leftarrow L_1, ..., L_n.$$

is deleted, the clause $a_{del}.$ is added to the program $P!$ where a_{del} is the deletion atom in the clause

$$a \leftarrow L_1, ..., L_n, not\ a_{del}.$$

in $P!$

This definition simulates deletion of a clause by the addition of a deletion atom. The addition of a clause is straightforward.

Definition 15.3 Whenever a clause:

$$a \leftarrow L_1, ..., L_n.$$

is added to the program P, the clause:

$$a \leftarrow L_1, ..., L_n, not\ a_{del}.$$

is added to $P!$ where a_{del} is a new deletion atom.

Now the transforming method from any program P to another program $P!$ is defined which can simulate deletion of the program P by the addition. Note that every time P is updated, $P!$ is also updated without deleting clauses in it.

Appendix B

Proof of Theorem 7.8

There does not exist an atom p such that $p \in K_1$ and *not* $p \in K_1$ because K_1 is empty.

Suppose that neither a positive literal p nor a negative literal *not* p is in K_i. Because no single derivation rule can add both p and *not* p at the same time, either $p \notin K_{i+1}$ or *not* $p \notin K_{i+1}$ holds.

Suppose that a positive literal p is in K_i but *not* p is not in K_i. A negative literal can be added to K_i only by the abductive derivation rule a4. However, a4 is not applicable because $p \in K_i$. Therefore *not* $p \notin K_{i+1}$.

Suppose that a negative literal *not* p is in K_i but p is not in K_i. A positive literal can be added to K_{i+1} only by the abductive derivation rules a1 or a3. Before *not* p was added to $K_j (j \leq i)$ by the abductive derivation rule a1, a consistency derivation was triggered where p was proved to be false and p was added to R_k $(k \leq j-1)$. Because $p \in R_i$, the abductive derivation rule a3 is not applicable. In order to apply a1 to add p to K_i, p must be a fact in P. But if p was a fact in P, p could not have been to be false in a consistency derivation. Therefore $p \notin K_{i+1}$.

Because for any i, $K_i \subseteq K_{i+1}$ holds, by induction, there does not exist an atom p such that $p \in K_t$ and *not* $p \in K_t$.

Proof of Theorem 7.10

Because the abductive derivation rules a2, a3, and a4 and the consistency derivation rules c3, c4, c5, and c6 are applied to abducibles, the effects of these rules under the programs P and Q are same.

If all the clauses in K_t are in Q, the effect of applying the abductive derivation rule a1 is the same under P and Q if the same clause is chosen.

If there does not exist a clause in Q which defines an atom in R_t other than those clauses in P, for every single consistency derivation from $(\emptyset, NG_i, K_i, R_i)$ to $(\emptyset, NG_{i+1}, K_{i+1}, R_{i+1})$ for

which c1 or c2 was applied, K_{i+1} and R_{i+1} are not affected by the change of the program if the same nand set and literal are chosen. Although, in this case, NG_{i+1} may be affected, the counterpart of NG_{i+1} under Q is a subset of NG_{i+1}. Therefore even under Q, every nand set which need to be resolved by a consistency derivation rule can be resolved in the same way as the given in complete proof.

Consequently, if there exists an abductive derivation from $(PG_t, \emptyset, K_t, R_t)$ to $(\emptyset, \emptyset, K_m, R_m)$ under Q, there exists a complete proof for the goals $L_1...L_n$ under the program Q from $(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$ to $(\emptyset, \emptyset, K2_x, R2_x)$, such that $K2_x \subseteq K_m$ and $R2_x \subseteq R_m$ if $K2_x$, $R2_x$ are constructed by the above method under Q. Because $K2_x \subseteq K_m$ and $R2_x \subseteq R_m$, $Q \cup \Delta_1 \vdash L_1 \wedge ... \wedge L_n$ holds as well.

Proof of Theorem 7.11

The effects of applying the consistency derivation rules c1 and c2 are the same under P and Q if the same nand set and literal are chosen.

In the proof of Theorem 7.10, it was shown that the effects of all the other abductive and consistency derivation rules are the same under P and Q.

Consequently, the incomplete proof IP is not affected by the change of the programs from P to Q.

Therefore the sequence $(\{L_1, ..., L_n\}, \emptyset, \emptyset, \emptyset)$ ... $(\emptyset, \emptyset, K_m, R_m)$ is a complete proof for the goals $L_1...L_n$ under Q.

Proof of Theorem 12.4

It suffices to show that the sequence S: $(\{L_1, ..., L_h\}, \emptyset, \emptyset, \emptyset)$ ... (PG_x, NG_x, K_x, R_x) , where (PG_x, NG_x, K_x, R_x) is the backtrack point, is an abductive derivation under the program Q.

Neither S nor the abductive derivations called by S are affected by the change of the program from P to Q because all the clauses used in abductive derivations are also available in Q.

Similarly no consistency derivations called by S are affected because all the clauses used in consistency derivations are also available in Q and these clauses are the only clauses which define the atoms in K_x (i.e. the chosen literals).

Therefore, the sequence S is a derivation under the program Q.

Proof of Theorem 12.6

All the selected literals other than $c_1, ..., c_n$ in abductive derivations can be resolved under Q in the same way as under P because all the chosen clauses used in abductive derivations are not removed.

The selected literal for any goal set other than $a_1, ..., a_m$ in consistency derivations can be resolved in the same way under Q as under P if no clause in P which defines the chosen literal is deleted from Q because the definition clauses of the selected literal are not changed.

Let L be the selected literal in the selected nand set N for any goal set in consistency derivations other than $a_1, ..., a_m$. Let R_P be the set of all the resolvents of N under P. If the same nand set and chosen literal are selected under Q, the set of all the resolvents R_Q will be a subset of R_P (**Lemma A**) because the set of all the definition clauses of L under Q is a subset of those under P. Although R_P needs to be resolved under Q in this theorem, if R_P can be resolved under Q, R_Q can also be resolved under Q because $R_Q \subseteq R_P$.

Consequently it suffices to show that $a_1, ..., a_n$ in addition to the literals in PG_i can be resolved to $\emptyset$ in an abductive derivation under Q adding further restrictions to $K_i \backslash DKC$ and R_i and that $\{b_1\}, ..., \{b_m\}$ can be resolved to $\emptyset$ in a consistency derivation under Q adding further restrictions.

Because in Lemma A, $R_Q \subseteq R_P$, resolving R_P under P will result in $K2_x \subseteq K_k$ and $R2_x \subseteq R_k$ if $K2_x$ and $R2_x$ are constructed by resolving R_Q under Q. Because $K2_x \subseteq K_k$ and $R2_x \subseteq R_k$, $Q \cup \Delta_1 \vdash L_1 \wedge ... \wedge L_u$ holds as well.

Proof of Corollary 12.9

Straightforward from Theorem 12.6