THE JOURNAL OF LOGIC PROGRAMMING

Computing abduction by using TMS with top-down expectation

Noboru Iwayama[a,*], Ken Satoh[b]

[a] *Fujitsu Laboratories Ltd., Nishiwaki 64, Ohkubo, Akashi 674-8555, Japan*

[b] *Division of Electronics and Information Engineering, Hokkaido University, Sapporo 060-8628, Japan*

Received 1 May 1998; received in revised form 1 April 1999; accepted 23 September 1999

Abstract

We present a method to compute abduction in logic programming. We translate an abductive framework into a normal logic program with integrity constraints and show the correspondence between generalized stable models and stable models for the translation of the abductive framework. Abductive explanations for an observation can be found from the stable models for the translated program by adding a special kind of integrity constraint for the observation. Then, we show a bottom-up procedure to compute stable models for a normal logic program with integrity constraints. The proposed procedure excludes the unnecessary construction of stable models on early stages of the procedure by checking integrity constraints during the construction and by deriving some facts from integrity constraints. Although a bottom-up procedure has the disadvantage of constructing stable models not related to an observation for computing abductive explanations in general, our procedure avoids the disadvantage by expecting which rule should be used for satisfaction of integrity constraints and starting bottom-up computation based on the expectation. This expectation is not only a technique to scope rule selection but also an indispensable part of our stable model construction because the expectation is done for dynamically generated constraints as well as the constraint for the observation.

Keywords: Abductive logic programming; Generalized stable models; Integrity constraint; Bottom-up computation

1. Introduction

We present a method of calculating abduction discussed in Refs. [11,20,21]. Recent researches have revealed that abduction plays an important role in artificial intelligence, as stated in Ref. [11]. Various researchers have studied abduction in terms

* Corresponding author. Tel.: +81-78-934-8248; fax: +81-78-934-3312.
E-mail address: iwayama@acm.org (N. Iwayama).

0743-1066/00/$ - see front matter
PII: S0743-1066(99)00076-X

of logic programming framework [11,20,21] (called *abductive logic programming*) and shown relationships with nonmonotonic reasoning framework such as negation as failure, assumption-based truth maintenance system and autoepistemic logic. Kakas et al. [22] survey the field of abductive logic programming.

In this paper, we give a method of computing abduction in logic programming. To do that, we first give a translation of an abductive framework to a normal logic program with integrity constraints and show that a stable model [15] for the translated logic program coincides with some generalized stable model for the abductive framework defined in Ref. [21]. Then, we provide a nondeterministic bottom-up procedure which calculates stable models for a normal logic program with integrity constraints. With the above translation, our procedure computes abduction.

In contrast to previous works, the proposed method is important in the following three points. First, it is enough to consider normal logic programs with integrity constraints in order to handle abductive frameworks. We show that a set of hypotheses used for the explanation defined in an abductive framework coincides with a part of the stable model for the translated logic program which satisfies a special integrity constraint representing an observation. Second, our method is correct for any abductive logic program because of correctness of our procedure proposed in this paper. Third, our procedure is designed so as to be suitable for query evaluation (especially for query for abductive observations), though the procedure computes models in bottom-up. In the following, we discuss the second and third points of arguments of this paper in detail.

Kakas and Mancarella [20] have provided an abductive proof procedure to calculate an explanation for a given observation in abductive logic programming. They extend Eshghi's top-down procedure for abduction [11] in order to manipulate arbitrary hypotheses. However, their procedure inherits a problem of Eshghi's procedure that correctness does not hold in general for logic programs with recursion as shown in Ref. [11, p. 251].

In Ref. [34], authors proposed a query evaluation procedure for abductive logic programming. The procedure in Ref. [34] can be regarded as an extension of the procedure of Kakas and Mancarella [20] by adding forward (or bottom-up) evaluation of rules and consistency check for implicit deletion. The procedure in Ref. [34] is correct for generalized stable model semantics when an abductive framework has at least one generalized stable model. In other words, the procedure in Ref. [34] is not correct for an abductive framework which has no generalized stable model.

The procedure proposed in this paper is based on a procedure calculating grounded extension [32] of TMS [8] and can be regarded as an extension of a well-founded bottom-up procedure for calculating stable model for logic programs *without* integrity constraints [13,31]. The procedure computes stable model semantics correctly for any normal logic program with integrity constraints. As a result, our method, the translation of abductive framework with the proposed procedure, is correct for any abductive framework whether the framework has generalized stable models or no model.

Although the proposed method in this paper is correct, one may suspect that a bottom-up procedure is not efficient for computing abductive explanations for an observation. We can compute hypotheses for an explanation by computing generalized stable models and then taking out hypotheses from the generalized stable models satisfying the observation. This is correct but apparently inefficient, because many of the generalized stable models not related to the observation might be constructed.

In the previous version [33] of this paper, authors have paid attention to integrity constraints to save search space as much as possible by excluding unsatisfiable stable models. We enhance a well-founded bottom-up procedure [13,31] by dynamically checking integrity constraints during the computation of stable models and by actively deriving some facts from integrity constraints. But, the procedure checks an integrity constraint corresponding to an abductive observation at the last stage of the procedure. This means that our procedure still suffers from construction of irrelevant models to an abductive observation.

In this paper, we overcome the inefficiency in the procedure proposed in Ref. [33]. We enhance our procedure to use information given in the observation by top-down expectation for the integrity constraint which is regarded as a goal in backward reasoning. The integrity constraint added for the observation is a sort of goal-like constraint. The idea of this enhancement is to search a rule which has the possibility of satisfying the integrity constraint and if such a rule does not exist, the procedure immediately fails.

Moreover, this enhancement contributes to efficiency even when there is no goal-like constraint explicitly presented in programs. Since constraints tried to be satisfied by the top-down expectation are dynamically generated during the bottom-up computation, the expectation is done for not only a constraint for an observation but also dynamically generated ones. This fact means that the enhancement is not only a technique to scope rule selection but also a part of our stable model construction.

The structure of the paper is as follows. In Section 2, we show definitions and properties of stable models. We define a translation of an abductive framework to a normal logic program with integrity constraints in Section 3. In Section 4, we provide a procedure which calculates a stable model for a normal logic program with integrity constraints. In Section 5, we consider an enhancement of the procedure by the top-down expectation. We mention related works and conclusion of the paper in the last two sections. As mentioned above, this paper is a revision and expansion of the paper in Ref. [33].

2. Stable model semantics

2.1. Preliminary definitions

We follow the definition in Refs. [20,21] but restrict ourselves to considering the propositional case. If we consider predicate case, we change it into a ground logic program by instantiating every variable to an element of Herbrand universe of considered logic program to obtain a propositional program.

Firstly, we define a normal logic program and integrity constraints.

Definition 1. Let A_i be propositional symbols. A normal logic program consists of (possibly countably infinite) rules of the form

$$A_0 \leftarrow A_1 \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n.$$

We call A_0 the head of a rule R (denoted as $head\_of(R)$) and $A_1, \ldots, A_n$ the body of the rule. The set of $\{head\_of(R)\}$, the set of propositions $A_1, \ldots, A_m$ in R, and the set of propositions $A_{m+1}, \ldots, A_n$ in R are denoted as $head(R)$, $pos(R)$ and $neg(R)$,

respectively. As a conceptual definition we consider the special rule *nil_rule* such that $head(nil\_rule) = pos(nil\_rule) = neg(nil\_rule) = \emptyset$.

We define integrity constraints as special rules.

Definition 2. Let A_i be propositions. A set of integrity constraints consists of (possibly countably infinite) rules of the form

$$\bot \leftarrow A_1 \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n.$$

In this definition, $\bot$ is the special propositional symbol which means contradiction. Since we write integrity constraints as rules, we do not have to distinguish constraints and rules. In the following, we call a normal logic program with integrity constraints as a normal logic program, or a logic program simply when it is obvious to distinguish that from the context. To be precise, we only consider a special form of integrity constraints whereas in Refs. [20,21] they allow any form of integrity constraints. However, those constraints can be translated into our form of integrity constraints.

We extend the definition of stable models in Ref. [15] for a normal logic program with integrity constraints as follows.

Definition 3. A program T be a normal logic program with integrity constraints. A stable model for a normal logic program with integrity constraints is a set of propositions M such that

1. M is equal to the least model (by set inclusion) of the positive program T^M where T^M is obtained by the following operation from T. We say that M is a stable model of T.
   (a) Deleting every rule R from T if some $n \in neg(R)$ is in M.
   (b) Deleting every negated atom in the remaining rules.
2. $\bot \notin M$.

This definition gives a stable model of T which satisfies all integrity constraints.

2.2. Groundedness of stable models

Stable models are shown to be related to nonmonotonic truth maintenance system (TMS) or justification-based TMS proposed by Doyle [8]. Because syntactic features of TMS resemble those of normal logic programming, we can show that the characteristics of TMS are borrowed to discuss the characteristics of normal logic programming. Elkan [10] showed that a stable model for a logic program without integrity constraints is equivalent to a grounded extension of TMS. Fages [13] showed the same result as Elkan independently. In this paper, we use the result by Elkan and Fages to consider our procedure calculating stable models. It is important to state the concept of grounded models because the concept implies a constructive definition of stable models.

The groundedness of TMS extensions is translated into the next definition. The definition is the same as Elkan's grounded model for logic programs except that

we consider logic programs with integrity constraints. The definition of grounded models provides us with the constructive definition of stable models.

Definition 4. Let T be a logic program with integrity constraints. A set of propositions M is a finite grounded model for T if the following are satisfied.
1. M is a model of T.
2. $\bot \notin M$.
3. M can be written as a sequence of propositions $\langle P_1, P_2, \ldots, P_n \rangle$ such that each P_j has at least one rule R_j such that $pos(R_j) \subseteq \{P_1, \ldots, P_{j-1}\}$, where $P_1, \ldots, P_{j-1}$ are the elements of the sequence up to $j-1$, $P_j = head\_of(R_j)$ and $neg(R_j) \cap M = \emptyset$. We say a sequence of such rules for every proposition in M, $\langle R_1, R_2, \ldots, R_n \rangle$, is a sequence of supporting rules for M.

We can prove the following corollary by extending Ref. [10, Theorem 3.8].

Corollary 5. *Let T be a logic program with integrity constraints. A set of propositions M is a finite grounded model for T if and only if M is a finite stable model for T.*

The definition of grounded model leads a procedure to compute stable model in constructive way. The procedure, denoted as *proc_S*, is shown in Figs. 1 and 2. We say that *proc_S outputs M with a sequence of rules $R_0, \ldots, R_n$* if $R_0, \ldots, R_n$ are selected in this order in *simple_select_rule* of *proc_S*, and *proc_S* outputs M after selecting these rules.

The procedure *proc_S* can compute stable models. In the following theorem, we show that outputs of *proc_S* are all stable models of a given logic program with integrity constraints.

Theorem 6.
1. *If proc_S outputs M, then M is a stable model for T.*
2. *If T is finite, then proc_S outputs all stable models by exhaustive search.*

begin

$i := 0$; $M_0 := \emptyset$;
$R := nil\_rule$; (remark: $head(nil\_rule) = \emptyset$)
do
  $i := i + 1$;
  $M_i := M_{i-1} \cup head(R)$;
  if there exists $R_k (0 \leq k \leq i-1)$ such that for some $N \in neg(R_k), N \in M_i$
  then fail;
  $R := simple\_select\_rule(M_i)$;
until $R = nil\_rule$;
if $\bot \in M_i$ **then fail else return** M_i;

end

Fig. 1. Procedure *proc_S*.

procedure *simple_select_rule*(M_i)
if there is a rule R in T satisfying the following conditions

(1) $head\_of(R) \notin M_i$,

(2) $pos(R) \subseteq M_i$,

(3) $neg(R) \cap M_i = \emptyset$.

then select the rule R and **return** R.
else return *nil_rule*.
endif

Fig. 2. Subprocedure *simple_select_rule*.

Proof.
1. If *proc_S* outputs M with a finite sequence of selected rules $R_0, \ldots, R_n$, then this sequence actually gives a sequence of supporting rules. We can show that M is a model of T and $\bot \notin M$. Therefore, M is a finite grounded model and, so, a finite stable model for T by Corollary 5.
2. Let T be a finite logic program. Suppose M is a stable model for T. Since T is a finite logic program, M is a finite stable model and therefore, a finite grounded model by Corollary 5. Then, there is a finite sequence of supporting rules $R_0, \ldots, R_n$. We can show that *proc_S* outputs M with this sequence of rules. Since all sequences of rules can be selected by exhaustive search, *proc_S* outputs all stable models. □

In Section 4, we will consider the procedure to compute stable models more efficiently than the procedure *proc_S*.

3. Translating abductive framework to logic program

In this section, we propose the translation of abductive frameworks into normal logic programs with integrity constraints. Firstly, we follow the definition of abductive framework in Refs. [20,21]. Since integrity constraints are represented as rules in the following definition, we do not need mention the constraints explicitly in the definition.

Definition 7. An abductive framework is a tuple $\langle T, A \rangle$ where
1. A is a (possibly countably infinite) set of propositional symbols called abducible propositions.
2. T is a normal logic program where no rule's head is equal to any element of A.

We follow the definition of generalized stable models and explanation with respect to $\langle T, A \rangle$ in Refs. [20,21].

Definition 8. Let $\langle T, A\rangle$ be an abductive framework and Δ be a subset of A. A generalized stable model $M(\Delta)$ of $\langle T, A\rangle$ is a stable model of $T \cup F(\Delta)$ where $F(\Delta) = \{p \leftarrow | p \in \Delta\}$.

Definition 9. Let $\langle T, A\rangle$ be an abductive framework, and q be a proposition called observation, and Δ be a subset of A. The proposition q has an abductive explanation with a set of hypotheses Δ if and only if there exists a generalized stable model $M(\Delta)$ such that $q \in M(\Delta)$.

At first sight, an abductive framework seems to extend logic programming by introducing abducibles, but it turns out that we can embed an abductive framework into a logic program with integrity constraints. We translate an abductive framework as follows.

Definition 10. Translation of abductive framework

Let $\langle T, A\rangle$ be an abductive framework.

1. For each abducible p in A, we introduce a new proposition $\tilde{p}$ which is not used in $\langle T, A\rangle$.
2. We add the following pair of rules in T for each abducible p in A

$$p \leftarrow not\,\tilde{p} \quad \text{and} \quad \tilde{p} \leftarrow not\,p.$$

We denote the set of added rules as $\Gamma(A)$, that is

$$\Gamma(A) = \{p \leftarrow not\,\tilde{p} \,|\, p \in A\} \cup \{\tilde{p} \leftarrow not\,p \,|\, p \in A\}.$$

3. We obtain a normal logic program $T \cup \Gamma(A)$ as the translation of the abductive framework $\langle T, A\rangle$.

The above pair of rules expresses that p and $\tilde{p}$ are mutually exclusive. So, $\tilde{p}$ intuitively means that p is not believed. The first rule $p \leftarrow not\,\tilde{p}$ is used to assume p and the second rule $\tilde{p} \leftarrow not\,p$ is used not to assume p. Especially, if we get a contradiction by assuming p, the latter rule is used to prevent the former rule from being used to assume p. In Ref. [31] Saccà and Zaniolo state that stable model semantics endows logic programming with the expressive power of don't-care nondeterminism. The above translation is regarded as the utilization of the nondeterminism in stable model semantics.

We have the following correspondence between abductive framework and its translation.

Theorem 11. *Let $\langle T, A\rangle$, $T \cup \Gamma(A)$, and Δ be an abductive framework, its translation, and a subset of A respectively. $M(\Delta)$ is a generalized stable model of $\langle T, A\rangle$ if and only if there exists a stable model M' for $T \cup \Gamma(A)$ such that $M' = M(\Delta) \cup \widetilde{V}$ where $\widetilde{V} = \{\tilde{p} \,|\, p \in (A - \Delta)\}$.*

Proof. See Appendix A. □

Example 12 (*Generalized stable models*). Consider the following logic program T with abducibles $A = \{a, b\}$:

$$p \leftarrow b \tag{1}$$
$$q \leftarrow a \tag{2}$$
$$r \leftarrow not\, b \tag{3}$$
$$\bot \leftarrow q, b \tag{4}$$
$$\bot \leftarrow not\, q, r \tag{5}$$

From the above abductive framework $\langle T, A\rangle$, we can get $M_1(\Delta_1) = \{b, p\}$ where $\Delta_1 = \{b\}$ and $M_2(\Delta_2) = \{a, q, r\}$ where $\Delta_2 = \{a\}$ as generalized stable models.

Translation from this abductive framework is as follows. We will add the following rules, $\Gamma(A)$ to the above logic program.

$$a \leftarrow not\, \tilde{a} \tag{6}$$
$$\tilde{a} \leftarrow not\, a \tag{7}$$
$$b \leftarrow not\, \tilde{b} \tag{8}$$
$$\tilde{b} \leftarrow not\, b \tag{9}$$

We can see that the following two sets of propositions are actually stable models of $T \cup \Gamma(A)$:

$$M_1' = M_1(\Delta_1) \cup \widetilde{\nabla_1} = \{b, p\} \cup \{\tilde{a}\} = \{b, p, \tilde{a}\}$$

and

$$M_2' = M_2(\Delta_2) \cup \widetilde{\nabla_2} = \{a, q, r\} \cup \{\tilde{b}\} = \{a, q, r, \tilde{b}\}.$$

We have another correspondence with respect to hypotheses used for explanation.

Corollary 13. *Let $\langle T, A\rangle$ be an abductive framework, $T \cup \Gamma(A)$ its translation, and q an observation. The observation q has an explanation with a set of hypotheses Δ if and only if there is a stable model M for $T \cup \Gamma(A) \cup \{\bot \leftarrow not\, q\}$ such that $\Delta = M \cap A$.*

Proof. See Appendix A. □

Example 14 (*Explanation*). Consider the abductive framework in Example 12. Suppose an observation q is given. This observation has the unique explanation with a set of hypotheses $\{a\}$. The following integrity constraint corresponding to the observation is added to the program $T \cup \Gamma(A)$:

$$\bot \leftarrow not\, q \tag{10}$$

$M_2' = \{a, q, r, \tilde{b}\}$ is the unique stable model for $T \cup \Gamma(A) \cup \{\bot \leftarrow not\, q\}$ and we can see that $M_2' \cap A = \{a\}$ is equal to the set of hypotheses.

4. Computing stable models for programs with integrity constrains

In this section, we give a nondeterministic bottom-up procedure to compute stable models for a logic program with integrity constraints. To combine the previous translation and the following procedure, we can calculate abduction.

There are previous works to deal with procedures calculating stable models. Saccà and Zaniolo proposed the backtracking fixpoint procedure in Ref. [31]. Fages considered a fixpoint semantics related to well-founded model semantics and stable model semantics in Ref. [13]. Fages mentioned a bottom-up procedure based on his fixpoint semantics. When considering only computation for normal logic programs without integrity constraints, procedures shown in Refs. [13,31] are more efficient than the procedure *proc_S* in Section 2.2, because the procedure in Refs. [13,31] keeps track of which propositions are assumed to be out of models. Our bottom-up procedure proposed in the below is same as the procedures in Refs. [13,31] on the point of keeping propositions not included in models.

From the definition of stable models, one might think that it is sufficient to use the procedure of Refs. [13,31] and remove every stable model which does not satisfy some integrity constraints in order to obtain all stable models. The procedure *proc_S* shown in Section 2.2 computes exactly in this way on integrity constraints. However, we may save search space if we can check integrity constraints during the process of constructing stable models. The following procedure performs not only such dynamic checking of integrity constraints but also active use of integrity constraints to derive some facts. Moreover, we can use integrity constraints introduced for a given observation actively to find hypotheses used in explanation for the observation.

There is one further point on integrity constraint that we must not ignore in our procedure. We have the cases in which rules are considered as integrity constraints besides rules' direct contribution to model construction (addition of heads to models). During model construction in our procedure, suppose that a proposition $p(= head\_(R))$ is decided to be out of belief for the rule R, or we decide that $not\, p$ is true. After that point, the rule R works as an integrity constraint C such that $head\_of(C) = \bot$, $pos(C) = pos(R)$ and $neg(C) = neg(R)$. This accelerates the model construction in our procedure, because this means that a new integrity constraint, which is not in an original program, is added to the program.

At first, in Fig. 3 we give a skeleton of the procedure to show how the procedure works. In the procedure, M_i expresses a set of propositions which are decided to be in

begin

$i := 0$; $M_0 := \emptyset$; $\widetilde{M}_0 := \{\bot\}$;
$R := nil\_rule$; (remark: $head(nil\_rule) = neg(nil\_rule) = \emptyset$)

do

i:=i+1;
$M_i, \widetilde{M}_i := propagate(M_{i-1} \cup head(R), \widetilde{M}_{i-1} \cup neg(R))$;
if $M_i \cap \widetilde{M}_i \neq \emptyset$ then **fail**;
$R := select\_rule(M_i, \widetilde{M}_i)$;

until $R = nil\_rule$;
return M_i;

end

Fig. 3. A procedure to compute a stable model (skeleton).

the belief set after selecting i rules by *select_rule* shown in Fig. 4. In subprocedure *select_rule*, simple integrity checking is incorporated into subprocedure *simple_select_rule* in Fig. 2. In the next section, subprocedure *select_rule* in Fig. 3 will be replaced to subprocedure *topdown_select* so as to exclude unnecessary model construction.

$\widetilde{M}_i$ expresses a set of propositions which are decided to be out of the belief set. We add the proposition $\bot$, representing contradiction, to $\widetilde{M}_0$ at the beginning of the procedure because all integrity constraints should be satisfied. If there is a conflict between M_i and $\widetilde{M}_i$ then M_i is not a possible candidate for a stable model. The procedure has a nondeterministic choice point in subprocedure *select_rule*. Therefore **fail** in the procedure expresses trying an alternative choice. Although we do not assume any specific search strategy in this procedure, the trial is returning to the most recent choice point under depth-first search strategy.

In the following subsection, we provide details of subprocedure *propagate*.

4.1. Subprocedure of propagation

Subprocedure *propagate* constructs a model candidate and checks integrity constraints actively. Subprocedure *propagate* in Fig. 5 performs the following three tasks:

1. *Bottom-up construction of the model* (by case 1): The body of a rule is satisfied in the current model candidate, and the head of the rule is not included either positively or negatively in the model candidate yet. So, the head is included in the model candidate.
2. *Dynamic checking of the integrity constraint* (by case 3): The body of a rule is satisfied in the current model candidate, but the head of the rule is decided to be out of the belief set. Since this means contradiction, the procedure fails and tries an alternative choice. The integrity constraints in original programs whose heads

procedure $select\_rule(M_i, \widetilde{M}_i)$
if there is a rule R in T satisfying the following conditions

(1) $head\_of(R) \notin M_i$,

(2) $pos(R) \subseteq M_i$,

(3) $neg(R) \cap M_i = \emptyset$.

then

  select the rule R
  if $head\_of(R) \in \widetilde{M}_i$ **then fail else return** R

else return nil_rule.
endif

Fig. 4. Subprocedure *select_rule*.

procedure *propagate*$(M_i, \widetilde{M}_i)$
begin

$k := 0$; $M_i^0 := M_i$; $\widetilde{M}_i^0 := \widetilde{M}_i$;

do

$k := k + 1$; $M_i^k := M_i^{k-1}$; $\widetilde{M}_i^k := \widetilde{M}_i^{k-1}$;

For every rule R in T

(1) if $head\_of(R) \notin M_i^{k-1}$, $pos(R) \subseteq M_i^{k-1}$, and $neg(R) \subseteq \widetilde{M}_i^{k-1}$ then $M_i^k := M_i^k \cup head(R)$;

(2) if $head\_of(R) \in \widetilde{M}_i^{k-1}$, $neg(R) \subseteq \widetilde{M}_i^{k-1}$, and $\exists p \in pos(R)$ s.t. $\{p\} = pos(R) - M_i^{k-1}$ then $\widetilde{M}_i^k := \widetilde{M}_i^k \cup \{p\}$;

(3) if $head\_of(R) \in \widetilde{M}_i^{k-1}$, $pos(R) \subseteq M_i^{k-1}$, and $neg(R) \subseteq \widetilde{M}_i^{k-1}$ then **fail**;

until $M_i^k = M_i^{k-1}$ **and** $\widetilde{M}_i^k = \widetilde{M}_i^{k-1}$;

return $M_i^k, \widetilde{M}_i^k$

end

Fig. 5. Subprocedure *propagate*.

are $\bot$ are also checked in this case, because $\bot$ is included in the $\widetilde{M}_0$ at the beginning of the procedure.

3. *Active use of the integrity constraints* which derive that *not* q is true from the integrity constraint $\bot \leftarrow q$. (by case 2): The propositions in the body of a rule are decided to be true except that one of the propositions (p) in the body is not decided yet to be true or false. In addition to this condition, the head of the rule is decided to be false. In this situation, suppose the proposition p be decided to be true, then the decision brings us contradiction. So we should decide that the proposition p is false in stable model, or that p is added to $\widetilde{M}$.

The sets M_i and $\widetilde{M}_i$ are equivalent to M_i and $\widetilde{M}_i$ in the procedure of Ref. [31, p. 215] except that in our procedure we check integrity constraints dynamically (the conflict checking in the main procedure and case 3 in subprocedure *propagate*) and $\widetilde{M}_i$ might increase by case 2 in subprocedure *propagate*.

4.2. Examples

We compare our procedure with the procedure of Ref. [31] with integrity constraint check afterwards. The procedure of Ref. [31] is essentially the same as the procedure *proc_S* described in Section 2.2. The following example shows the difference.

Example 15 (*Difference of two procedures*). Consider the following program:

$$p \leftarrow q \tag{1}$$

$$r \leftarrow not\, q \tag{2}$$

$$q \leftarrow not\, r \tag{3}$$

$$\bot \leftarrow p \tag{4}$$

The procedure of Ref. [31] produces stable models $\{p, q\}$ and $\{r\}$ for a logic program of (1), (2) and (3). So, this process has a nondeterminism of producing two stable models. Then, we discard $\{p, q\}$ because this model does not satisfy the integrity constraint (4).

On the other hand, the execution of our procedure is as follows:

0. $M_0 = \{r\}$, $\widetilde{M}_0 = \{\bot, p, q\}$, because from (4), p must be in $\widetilde{M}_0$ by case 2 in propagate, and from (1), q must be in $\widetilde{M}_0$ by case 2 in propagate, and from (2), r must be in M_0 by case 1 in propagate.
1. Since *select_rule* returns *nil_rule*, M_0 is returned.

Therefore, in this example, we have calculated the stable model deterministically in our procedure. Note that, in this execution, the integrity constraint (4) is used to derive that p is out of belief and the rule (1) is used to derive that q is out of belief. So, this example shows an active usage of integrity constraints and a top-down propagation of a disbelieved atom in our procedure.

We can calculate abduction by combining the translation from an abductive framework into a logic program with integrity constraints and the above procedure to compute stable models for the translated logic program with integrity constraints.

Example 16 (*Combination of translation and bottom-up procedure*). We calculate abductive explanation for (1)–(9) in Example 12 and the observation (10) in Example 14. We show how M_i and $\widetilde{M}_i$ are constructed for all combinations for selections of the rules in the above procedure.

Selection 1.

0. $M_0 = \emptyset$, $\widetilde{M}_0 = \{\bot\}$.
1. Select rule (3). Then, $M_1 = \{r, \tilde{b}\}$, $\widetilde{M}_1 = \{\bot, b\}$.
2. Select rule (6). Then, $M_2 = \{r, \tilde{b}, a, q\}$, $\widetilde{M}_2 = \{\bot, b, \tilde{a}\}$.
3. Since there is no selected rule, M_2 is returned.

Selection 2.

0. $M_0 = \emptyset$, $\widetilde{M}_0 = \{\bot\}$.
1. Select rule (3). Then, $M_1 = \{r, \tilde{b}\}$, $\widetilde{M}_1 = \{\bot, b\}$.
2. Select rule (7). Then, $M_2 = \{r, \tilde{b}, \tilde{a}\}$, $\widetilde{M}_2 = \{\bot, b, a\}$.
3. Although there is no selected rule, M_2 does not satisfy integrity constraint (10). So, this process fails.

Selection 3.

0. $M_0 = \emptyset$, $\widetilde{M}_0 = \{\bot\}$.
1. Select rule (6). Then, $M_1 = \{a, q, r, \tilde{b}, r\}$, $\widetilde{M}_1 = \{\bot, \tilde{a}, b\}$.
2. Since there is no selected rule, M_1 is returned.

Selection 4.

0. $M_0 = \emptyset$, $\widetilde{M}_0 = \{\bot\}$.
1. Select rule (7). Then, $M_1 = \{\tilde{a}\}$, $\widetilde{M}_1 = \{\bot, a\}$.
2. Select rule (3). Then, $M_2 = \{\tilde{a}, r, \tilde{b}\}$, $\widetilde{M}_2 = \{\bot, a, b\}$.
3. Although there is no selected rule, M_2 does not satisfy integrity constraint (5). So, this process fails.

Selection 5.
0. $M_0 = \emptyset,\ \widetilde{M}_0 = \{\bot\}$.
1. Select rule (7). Then, $M_1 = \{\tilde{a}\},\ \widetilde{M}_1 = \{\bot, a\}$.
2. Select rule (8). Then, $M_2 = \{\tilde{a}, b, p\},\ \widetilde{M}_2 = \{\bot, a, \tilde{b}, q, r\}$.
3. Although there is no selected rule, M_2 does not satisfy integrity constraint (10). So, this process fails.

Selection 6.
0. $M_0 = \emptyset,\ \widetilde{M}_0 = \{\bot\}$.
1. Select rule (7). Then, $M_1 = \{\tilde{a}\},\ \widetilde{M}_1 = \{\bot, a\}$.
2. Select rule (9). Then, $M_2 = \{\tilde{a}, \tilde{b}, r\},\ \widetilde{M}_2 = \{\bot, a, b\}$.
3. Although there is no selected rule, M_2 does not satisfy integrity constraint (5). So, this process fails.

Selection 7.
0. $M_0 = \emptyset,\ \widetilde{M}_0 = \{\bot\}$.
1. Select rule (8). Then, $M_1 = \{b, p, \tilde{a}\},\ \widetilde{M}_1 = \{\bot, \tilde{b}, q, a, r\}$.
2. Although there is no selected rule, M_1 does not satisfy integrity constraint (10). So, this process fails.

Selection 8.
0. $M_0 = \emptyset,\ \widetilde{M}_0 = \{\bot\}$.
1. Select rule (9). Then, $M_1 = \{\tilde{b}, r\},\ \widetilde{M}_1 = \{\bot, b\}$.
2. Select rule (6). Then, $M_2 = \{\tilde{b}, r, a, q\},\ \widetilde{M}_2 = \{\bot, b, \tilde{a}\}$.
3. Since there is no selected rule, M_2 is returned.

Selection 9.
0. $M_0 = \emptyset,\ \widetilde{M}_0 = \{\bot\}$.
1. Select rule (9). Then, $M_1 = \{\tilde{b}, r\},\ \widetilde{M}_1 = \{\bot, b\}$.
2. Select rule (7). Then, $M_2 = \{\tilde{b}, r, \tilde{a}\},\ \widetilde{M}_2 = \{\bot, b, a\}$.
3. Although there is no selected rule, M_2 does not satisfy integrity constraint (5). So, this process fails.

So by exhaustive search, we find all stable models for (1)–(9) and (10). In this example, the program has only one stable model, that is $\{a, q, r, \tilde{b}\}$.

5. Incorporating top-down expectation

As shown in the previous example, the procedure shown above generates irrelevant models to the integrity constraint corresponding to an abductive observation. In this section, we consider how to select a rule in order to drive the bottom-up model construction so as to exclude the irrelevant cases.

5.1. Idea of top-down expectation

To exclude irrelevant models, there are cases where we should select a specific rule. For the integrity constraint, $\bot \leftarrow not\, p$, for example, which is introduced to find the hypotheses in abductive explanation in the previous section, we know that p must be in the models. Suppose p is not in any stable model. This means that there is no stable model satisfying the integrity constraint. So first of all, we should select a rule

which directly derives p. If such a rule does not exist, we should select a rule which has a chance of deriving p. Therefore, it is important to consider the integrity constraint of the form of $\bot \leftarrow not\, p$ at the early stage of the procedure, especially for computing an abductive explanation for an observation.

To explain the possibility and chance to derive p, we consider the following example:

$$p \leftarrow q, r, not\, s \tag{1}$$

$$q \leftarrow not\, t \tag{2}$$

$$r \leftarrow not\, u \tag{3}$$

Given the integrity constraint, $\bot \leftarrow not\, p$, p must be in all the models of the above example. Because only rule (1) has p as its head, rule (1) must be used to derive p. In order for p to be in the models, q and r must also be in the models by rule (1). We can derive q from rule (2) if we can assume that t is not in the models, so rule (2) has a possibility to derive q. In a similar way, we find that rule (3) also has a possibility to derive r. Both rules (2) and (3) have therefore the chance to derive p. In this way, we can find rules with possibility or chance to derive p in a top-down manner from an integrity constraint of the form $\bot \leftarrow not\, p$.

Although one might say that this enhancement is just a technique to scope rule selection, which is known well in the field of theorem proving, the enhancement plays an important role in our stable model construction. In fact, this enhancement is an equivalent technique to a relevance check [28] which is added to Manthey and Bry's model generation theorem prover SATCHMO [23] for first-order predicate calculus as we show in Section 6.5. However, we argue that the top-down expectation is an important device in our whole stable model construction procedure.

In addition to a given constraint for observation, the expectation is done for other constraints. Let us consider the rule $p \leftarrow not\, q$. Suppose, during the computation, the proposition p has been decided to be not in the model. After this point, we treat the rule as an integrity constraint $\bot \leftarrow not\, q$ because q must be in the model. This means that we can do top-down expectation for this "dynamically" generated constraint. In other words, the expectation contributes to efficiency even when there is no goal-like constraint explicitly presented in programs.

5.2. Implementing top-down expectation

In the following, we describe technical details of the top-down expectation.

To check whether there is a selectable rule and to decide which rule should be selected, we modify the procedure in Fig. 3. We replace subprocedure *select_rule* with subprocedure *topdown_select* (Fig. 6). Subprocedure *topdown_select* plays the same role to select one rule among selectable rules as subprocedure *select_rule*. But in *topdown_select* the selection is done on the basis of top-down information, instead of a random selection in *select_rule*.

The purpose of *topdown_select* is to find a starting point at which subprocedure *topdown_check* performs backward reasoning (Fig. 7). The starting point is a proposition p for a form of integrity constraint like $\bot \leftarrow not\, p$. Initially *topdown_select* checks whether there is this type of integrity constraint for which top-down expecta-

procedure *topdown_select*$(M_i, \widetilde{M}_i)$
if there exists R in T such that

(1) $neg(R) - \widetilde{M}_i \neq \emptyset$,

(2) $head\_of(R) \in \widetilde{M}_i$,

(3) $pos(R) \subseteq M_i$ and

(4) $neg(R) \cap M_i = \emptyset$.

then

select a proposition pr from $neg(R) - \widetilde{M}_i$;
return *topdown_check*$(M_i, \widetilde{M}_i, pr)$

else

if there is a rule R in T satisfying the following conditions

(1) $head\_of(R) \notin M_i$,
(2) $pos(R) \subseteq M_i$,
(3) $neg(R) \cap M_i = \emptyset$.

then select the rule R
if $head\_of(R) \in \widetilde{M}_i$ **then fail else return** R
else return *nil_rule*.

endif

Fig. 6. Subprocedure *topdown_select*.

tion is performed. The check is done not only for integrity constraints but also rules whose heads are already in $\widetilde{M}_i$, because such rules play same role as integrity constraints. If there is not such an integrity or rule (**else** part), top-down expectation is not performed and we must select a rule with no clue (in exactly same way as *select_rule*). Otherwise, we perform top-down expectation (**then** part).

In the part **then** of *topdown_select*, we select the starting point from propositions in an integrity (or rule) not from propositions of all integrity constraints (or rules) satisfying the **if** condition. It is enough to select the starting point of backward reasoning from propositions in one integrity, because the failure of all possibilities of backward reasoning from the starting point means that there is no model in which the integrity constraint is satisfied.

Subprocedure *topdown_check* performs backward reasoning from the proposition found in *topdown_select* in order to find a rule starting the bottom-up model construction in our whole procedure. In *topdown_check*, by M_t and $\widetilde{M}_t$, a rule is selected which is consistent with the rules previously selected during top-down expectation. Moreover M_u is used to exclude cyclic derivations. We also exclude local cycles at the **if** condition 2.

procedure *topdown_check*$(M, \widetilde{M}, p)$
$j := 0$;
$M_t := \widetilde{M}_t := M_u := \emptyset$;
$Pos := \{p\}$;
Label:
if for every proposition $pr \in Pos$, there is a rule R in T which satisfies the following conditions

(1) $head\_of(R) = pr$,

(2) $head\_of(R) \notin pos(R)$, $head(R)\_of \notin neg(R)$,

(3) $pos(R) \cap \widetilde{M} = pos(R) \cap \widetilde{M}_t = pos(R) \cap M_u = \emptyset$,

(4) $neg(R) \cap M = neg(R) \cap M_t = \emptyset$.

then

$pr_j :=$ an arbitrary proposition in Pos;
$Rset := \{R | R$ in T satisfies the above conditions and $head\_of(R) = pr_j\}$
select a rule R_j from $Rset$;
$Pos := pos(R_j) - M$;
if $Pos = \emptyset$ **then return** R_j
else
$M_t := M_t \cup pos(R_j)$;
$\widetilde{M}_t := \widetilde{M}_t \cup neg(R_j)$;
$M_u := M_u \cup head(R_j)$;
$j := j + 1$;
goto Label
endif
else fail
endif

Fig. 7. Subprocedure *topdown_check*.

In the part **then** of *topdown_check*, we commit one proposition in *Pos* and then select a rule to be able to derive the committed proposition. That is because we do not want to construct a complete proof tree for the starting point (goal p) of backward reasoning. We should notice that it is not necessary to select other propositions from *Pos* than pr_j. If all the alternatives to derive pr_j fail, goal p of *topdown_check* cannot be derived even though other propositions in *Pos* were proved.

Our procedure computes mainly in a bottom-up manner. Top-down expectation is only 'expectation', since procedure *topdown_check* does not construct a model but provides some clue in order to begin computation of procedure *propagate*.

We denote the procedure in Fig. 3 as *proc_O*, the procedure, in which subprocedure *select_rule* in *proc_O* is replaced with *topdown_select*, as *proc_N* respectively. As *proc_S* in Section 2.2, we say that *proc_O* (*proc_N*) *outputs M with a sequence of rules* $R_0, \ldots, R_n$ if $R_0, \ldots, R_n$ are selected in this order in *select_rule* (*topdown_select*) of *proc_O* (*proc_N*) and case 1 in *propagate*, and *proc_O* (*proc_N*) outputs M after selecting these rules.

In the next theorem, we show that models generated by *proc_O* (*proc_N*) are equivalent to models generated by *proc_S* presented in Section 2.2.

Theorem 17.
1. *If proc_O* (*proc_N*) *outputs M with a sequence of rules then proc_S outputs M with the same sequence of rules*
2. *Let T be finite. If proc_S outputs M with a sequence of rules then there exists a sequence of rules with which proc_O* (*proc_N*) *outputs M.*

Proof. See Appendix A. □

We can show that *proc_O* (*proc_N*) returns every stable model by an appropriate selection of rules, and it is complete for the finite propositional case.

Corollary 18. *Let T be a normal logic program with integrity constraints.*
1. *If proc_O* (*proc_N*) *outputs M, then M is a stable model for T.*
2. *If T is finite, then proc_O* (*proc_N*) *outputs all stable models by an exhaustive search.*

Proof. We have already showed that outputs by *proc_S* are actually all stable models of a given logic program (Theorem 6). The corollary is proved by Theorems 6 and 17. □

In the following example, we show how efficiently our procedure computes abductive explanations.

Example 19 (*Combination of translation and bottom-up procedure with top-down expectation*). We consider the same program in Example 16, which is the translation of an abductive framework and an observation. The execution of *proc_N* for the program is as follows:

0. $M_0 = \emptyset, \widetilde{M}_0 = \{\bot\}$.
1. In *topdown_select* an integrity constraint (10) satisfies the **if** condition as a candidate of starting point to reason backwards in *topdown_check*. Select rule (2) and (6) in *topdown_check*$(M_0, \widetilde{M}_0, q)$. Then, rule (6) is returned by *topdown_check* since rule (6) has no positive proposition. So, by *propagate*, $M_1 = \{a, q, r, \tilde{b}\}$, $\widetilde{M}_1 = \{\bot, \tilde{a}, b\}$.
2. Since *topdown_select* returns *nil_rule*, M_1 is returned.

So, we get an abductive explanation $M_1 \cap A = \{a\}$ by the correspondence proved in Section 3. Note that we deterministically compute the model at each choice point.

If we did not consider the top-down expectation, then we would have nine alternatives for selecting rules as shown in Example 16. Among the nine alternatives, there is the same rule selection (selection 3 in Example 16) as above. However, six of the other alternatives cannot provide any explanation for the observation q because the alternatives do not select rule (6) which is necessary to derive q. The other two alternatives (selection 1 and 8 in Example 16) result in the same model as the above computation but first select the irrelevant rules for q. This example makes clear that we can reduce the amount of backtracking thanks to top-down expectation of integrity constraints.

Table 1
Performance for the 3-coloring problem of ladder graphs

Vertices	Top-down		No top-down	
	Models	s	Models	s
4	18	0.10	408	1.19
6	54	0.42	27168	77.28
8	162	1.55		
10	486	5.37		

5.3. Experimental results

We have implemented our procedure in ordinary Prolog and KL1 (a concurrent logic programming language). The detail of the KL1 implementation is shown in Ref. [32]. In the following, we consider the Prolog implementation of our procedure.[1]

We show our experimental results in order to examine the effect of top-down expectation. The 3-coloring problem for ladder-graphs is used as test cases. As shown in Ref. [24], the 3-coloring problem is translated from a graph into a logic program in the following way. For each vertex a with neighbors $p_1, \ldots, p_j$ and each color $i \in 0, 1, 2$, the program includes the rule:

$$color(a, i) \leftarrow not\, color(p_i, i), \ldots, not\, color(p_j, i), \\ not\, color(a, i + 1 \bmod 3), not\, color(a, i + 2 \bmod 3)$$

and for each vertex a the program includes the integrity constraint:

$$\bot \leftarrow not\, color(a, 0), not\, color(a, 1), not\, color(a, 2).$$

The number of rules (and constraints) in the program is $4V$ if V is the number of vertices.

To examine the effect of top-down expectation, all stable models for the above program, which correspond to all possible colorings of the coloring problem, are computed. The computation is done by an exhaustive search of rule selection both for the procedures with and without top-down expectation.

The results are shown in Table 1. For each size of ladder graph, number of stable models computed and the time (seconds) to find all stable models are shown by the procedure with and without top-down expectation. The experiments were run on a PC (133 MHz Pentium processor, Linux 2.0.35, 48 MB of memory), and on SWI-Prolog (Version 3.2.2)[2] with default memory asignment. The timing is obtained by "time" meta call and "cputime" in SWI-prolog.

We have found all stable models (colorings) with no duplication by the procedure with top-down expectation, because in each call of *topdown_check* one rule is selectable in the program. On the other hand, without top-down expectation, same models are repeatedly generated, because all enumeration of selectable rules should be computed.

[1] The codes of the procedure and test case generator are available from http://www.inagaki.nuie.nagoya-u.ac.jp/person/iwayama/lp/.

[2] SWI-Prolog is available from http://swi.psy.uva.nl/projects/SWI-Prolog/.

Table 2
Comparison with SLG and sm

Vertices	Top-down	No top-down	SLG	sm
10	0.11	0.12	0.72	0.03
20	0.48	0.46	132.89	0.04
100	13.83	12.90		0.24

To compare our procedure with other procedures (SLG [3] and sm [25]), we show the time (seconds) to find first stable model for the 3-coloring problem in Table 2. The results of SLG and sm are cited from Ref. [25].

6. Related work

6.1. TMS

The procedure proposed in this paper is related to a procedure computing Doyle's TMS (justification-based Truth Maintenance System) [8] as stated in Section 2.2. In this paper, by generalizing Elkan's results [10] of the relationship between TMS and logic programming we modify a procedure for TMS to a procedure to compute stable models for a logic program *with* integrity constraints. There have been a lot of researches on semantics of Doyle's TMS [10,14,19,26,30]. However, none of these works except [14] considers integrity constraints (nogoods in TMS terminology) explicitly in the definition of TMS. In this connection, the algorithm in Ref. [19] is also related, but it does not consider nogoods explicitly. In Ref. [32], we have given a procedure which computes a grounded extension of TMS including nogoods.

Giordano and Martelli [16] have given a translation of a set of TMS justifications with integrity constraints to another set of justifications without integrity constraints to produce all stable models including stable models obtained by dependency-directed backtracking (DDB). Although this work is important in its own right to give a semantics for DDB of Doyle's TMS, this semantics conflicts with the original usage of integrity constraints in deductive databases, that is, checking integrity violation by updates. The problem is that even if an update is violated by the current integrity constraints, we might get other consistent states by performing DDB and therefore, we might not be able to detect a violation of the updates.

6.2. Computation of stable models

There are many works to show the methods computing stable models. Among the previous works, in Refs. [13,31] a well-founded bottom-up procedure for calculating stable models for logic programs *without* integrity constraints is provided. Our procedure can be regarded as an extension of the bottom-up procedure provided in Refs. [13,31]. The point of the extension is that our procedure deals with integrity constraints and utilizes the integrity constraints to reduce search spaces, especially for abduction by top-down expectation.

Eshghi [12] has given an algorithm using ATMS and a filtering mechanism to generate stable models from labels in ATMS. However, he considers only logic programs without integrity constraints.

In recent years, new directions for computing stable model semantics are proposed. Rao et al. [29] have developed a logic programming system XSB. Among its features, it should be noted that XSB evaluates programs according to well-founded semantics and provides a basis for computing partial stable model semantics [13]. While XSB does not support computation of partial stable model semantics directly, we can find partial stable models from results of XSB's computation.

Cholewiński et al. [4] proposed a reasoning system DeReS for Reiter's default logic. As a special case of default logic, DeReS computes stable model semantics for normal logic programs. DeReS has a technique that eliminates the need for some global consistency checks, because in DeReS, defaults in a theory are divided into strata and extensions of the original default theory are constructed by linking extensions of the strata [5]. This technique shows substantial speedup in computation.

Niemelä and Simons [24,25] developed a system for computing well-founded and stable model semantics for range-restricted function-free normal programs. The system, called Smodels, contains three components of expand, test and selection of proposition. In this sense, the system Smodels is closely related to our procedure because our procedure has the three corresponding components (*propagate*, consistency check, and *select_rule*) to those of Smodel. We distinguish Smodels to our procedure in the following two points. First, in subprocedure *propagate* we make the ordinary deductive closure stronger by considering integrity constraints like $\bot \leftarrow p$, while a Fitting operator is used in Smodels. Second, we divide cases according to applicable rules in *select_rule*, while the case splitting in Smodels is based on a proposition. In addition to assumption of a negative literal ($not\, p$), it is a substantially different point of Smodels from our procedure to assume a positive literal (p) and to check whether the assumed positive literal be able to be derived.

6.3. Translation of abductive logic programming

In this paper, we present a method of calculating abduction by translating an abductive framework into a logic program with integrity constraints and computing stable models for the program. Although there are previous works based on the transformation of abductive logic programs into other formalisms, the proposal by authors in the preliminary version [33] of this paper is, to the best of authors' knowledge, the first attempt.

Toni and Kowalski [35] are related to our approach deeply, because the translation in this paper is used as a part of the transformation in Ref. [35]. They show a transformation of abductive logic programs into normal logic programs without integrity constraints. They deal with default abducibles and non-default abducibles. Our translation is used for their transformation for default abducibles. The transformation is correct and complete with respect to many semantics formulated in an argumentation framework discussed in Refs. [1,2].

Inoue et al. [17] proposed a transformation of normal (and extended) logic programs into disjunctive logic programs, in which the fixpoints of the disjunctive programs correspond to stable models (answer sets) for the original programs. Inoue and Sakama [18] show a transformation of abductive logic programs into disjunctive logic programs based on the transformation proposed in Ref. [17] and the general-

ized stable models are captured in the fixpoints of the transformed disjunctive programs. The fixpoint of a transformed disjunctive program is obtained without nondeterministic choices. Instead of that, the fixpoint manages nondeterminism (discussed in Ref. [31]) in (generalized) stable models because the fixpoint consists of models which are constructed by branching models when dealing negation by failure literals and abducibles. Based on the characterization in Refs. [17,18], the procedures computing (generalized) stable models are implemented on bottom-up model generation theorem provers.

6.4. Top-down procedures for abductive logic programming

Our procedure proposed here mainly computes in bottom-up manner and uses top-down technique in *topdown_check* to find some clue starting the bottom-up computation. Procedures in Refs. [17,18] are bottom-up, or forward reasoning. From the view point of computing abductive logic programming in backward reasoning, there are related works to the proposed method.

Kakas and Mancarella [20] have extended Eshghi's top-down procedure [11] for abduction so that arbitrary abducibles can be used. Although their procedure is correct for a certain class of logic programs in stable model semantics,[3] they show that their procedure is suitable for a truth maintenance mechanism to manipulate consistent explanations for a series of observations which can be regarded as a non-monotonic extension of ATMS.

Authors have proposed a query evaluation method for abductive logic programming in Ref. [34]. The procedure in Ref. [34] can be regarded as an extension of the procedure of Kakas and Mancarella [20] by adding forward (or bottom-up) evaluation of rules and consistency check for implicit deletion. The procedure in Ref. [34] can be regarded as blending forward reasoning with backward reasoning. The procedure in Ref. [34] is correct for generalized stable model semantics when the program has at least one generalized stable model. In contrast to these previous works, the method proposed in this paper is correct for any program as mentioned in Section 1.

Denecker and De Schreye present an extension of SLDNF resolution for abductive logic programs in Refs. [6,7]. The procedure is correct under their 3-valued completion semantics. Their proof procedure, SLDNFA, solves the floundering abduction problems: non-ground abducibles can be selected, whereas we consider only ground abducibles. Since the procedure proposed in Refs. [17,18,34] deals with range-restricted and function free programs, we do not need consider non-ground abducibles in the procedures.

6.5. Forward reasoning enhanced by backward reasoning

We have two objectives to incorporate top-down expectation; one superficial reason is to search a rule which has the possibility of satisfying integrity constraints, and another more important one is to make use of dynamically generated top-down in-

[3] The top-down procedure by Eshghi and Kowalski [11] is sound with the preferential semantics by Dung [9].

formation in order to accelerate model construction. As we have suggested in Section 6.5, the top-down expectation is an equivalent technique to a relevance check [28] which is added to Manthey and Bry's model generation theorem prover SATCHMO [23] for first-order predicate calculus.

The theorem prover SATCHMO generates model candidates by forward reasoning to check satisfiability of formulas. When any SLD proof to yield contradiction under a current model candidate fails, SATCHMO splits and expands the model candidate by applying a disjunctive rule applicable under the current model candidate. At this stage, there might be possibilities to choose an irrelevant rule to proof for contradiction. The irrelevant selection of rule results in an explosion of model candidates.

Ramsay has enhanced the original SATCHMO so as to choose a disjunctive rule among rules which should contribute some proof for contradiction. Besides the proof for contradiction is done backward reasoning (SLD resolution on Prolog), Ramsay's enhanced SATCHMO checks whether the sub-goals failed in the SLD proof are unifiable with the heads in disjunctive model generation rules. This unification is called "relevance check". When the enhanced SATCHMO expands a model candidate, a disjunctive rule applied for the expansion is chosen among the rules passing relevance check. The relevance check, the rule selection based on the information given in backward reasoning, has an equivalent effect to our top-down expectation for rule selection to start bottom-up construction of stable models.

In addition to the relevance check, the magic set technique in Ref. [27] seems to be related to our procedures in this paper and Ref. [34] as suggested in Ref. [13]. Although our procedure in this paper uses a kind of backward reasoning at *topdown_check*, the reasoning is only for obtaining clues to start the bottom-up computation and not a complete reasoning because the reasoning does not yield refutation. By means of the magic set transformation, the magic rules are added to the original program to simulate backward reasoning in forward reasoning. However, our procedures both in this paper and Ref. [34] do not simulate backward reasoning but perform a kind of backward reasoning directly.

7. Conclusion

In this paper, we present a method of calculating abduction in logic programming by translating an abductive framework into a logic program with integrity constraints and computing stable models for the program. The following three points are particularly important from the viewpoint of computation. First, it is enough to consider normal logic programs with integrity constraints in order to handle abductive frameworks. By our translation, we are able to discuss abductive logic programming without treating abducibles explicitly. Second, our method is correct for any abductive logic program because of correctness of the procedure proposed in this paper. Third, our procedure is designed so as to be suitable for query evaluation (especially for query for abductive observations), though the procedure computes models in bottom-up. The comparison between our method and previous works in computational tractability is needed as the future consideration.

Acknowledgements

We thank anonymous referees and editors for helpful suggestions. This work is an expansion of a preliminary paper [33]. The preliminary work was done while authors were at the Institute for New Generation Computer Technology (ICOT). The expansion was started when the authors were at Fujitsu Laboratories. This work was part of the Ph.D. program of Noboru Iwayama at Nagoya University. We are indebted to Yasuyoshi Inagaki, Katsuhiko Toyama, Junichi Tanahashi, Tetsuji Morishita and Masahiro Matsuda for their support.

Appendix A

Proof of Theorem 11. We first prove the following lemma.

Lemma 20. *Let $\langle T, A\rangle$ be an abductive framework and $T' = T \cup \Gamma(A)$ and Δ be a subset of A. Let $M(\Delta)$ be a subset of propositions used in $\langle T, A\rangle$ such that $M(\Delta) \cap A = \Delta$. Let M' be $M(\Delta) \cup \widetilde{\overline{V}}$. Then,*

$$\min\left(T'^{M'}\right) = \min\left((T \cup F(\Delta))^{M(\Delta)}\right) \cup \widetilde{\overline{V}},$$

where $\min(\Upsilon)$ *means the minimal model of a positive logic program* Υ.

Proof. $\min\left(T'^{M'}\right) = \min((T \cup \Gamma(A))^{M'}) = \min(T^{M'} \cup \Gamma(A)^{M'})$. Since T does not contain symbols in $\widetilde{\overline{V}}$, $T^{M'} = T^{M(\Delta)}$.

And since $\Gamma(A)$ contains only symbols in $\Delta \cup \widetilde{\overline{V}}$, $\Gamma(A)^{M'} = \Gamma(A)^{\Delta \cup \widetilde{\overline{V}}}$.

For every abducible $p \in A$ and for every pair of rules in $\Gamma(A)$

if $p \in \Delta$ then $\{p \leftarrow not\,\tilde{p}, \tilde{p} \leftarrow not\,p\}^{\Delta \cup \widetilde{\overline{V}}} = \{p \leftarrow\}$

else if $p \notin \Delta$, that is, $\tilde{p} \in \widetilde{\overline{V}}$

then $\{p \leftarrow not\,\tilde{p}, \tilde{p} \leftarrow not\,p\}^{\Delta \cup \widetilde{\overline{V}}} = \{\tilde{p} \leftarrow\}$.

Therefore, $\Gamma(A)^{\Delta \cup \widetilde{\overline{V}}} = F(\Delta) \cup F(\widetilde{\overline{V}})$, where $F(\widetilde{\overline{V}}) = \{\tilde{p} \leftarrow \mid \tilde{p} \in \widetilde{\overline{V}}\}$. Thus

$$\begin{aligned}\min\left(T^{M'} \cup \Gamma(A)^{M'}\right) &= \min\left(T^{M(\Delta)} \cup F(\Delta) \cup F(\widetilde{\overline{V}})\right)\\ &= \min\left((T \cup F(\Delta))^{M(\Delta)} \cup F(\widetilde{\overline{V}})\right) \text{ since } F(\Delta) = (F(\Delta))^{M(\Delta)}\\ &= \min\left((T \cup F(\Delta))^{M(\Delta)}\right) \cup \widetilde{\overline{V}}\end{aligned}$$

since no common symbols in $T \cup F(\Delta)$ and $\widetilde{\overline{V}}$. □

Now we prove Theorem 11.

(1) Assume $M(\Delta) = \min\left((T \cup F(\Delta))^{M(\Delta)}\right)$ and $\bot \notin M(\Delta)$.

$$\begin{aligned}\min\left(T'^{M'}\right) &= \min\left((T \cup F(\Delta))^{M(\Delta)}\right) \cup \widetilde{\overline{V}} \quad \text{(by Lemma 20)},\\ &= M(\Delta) \cup \widetilde{\overline{V}} \quad \text{(by the assumption)},\\ &= M'\end{aligned}$$

This means that M' is a stable model of T' because $\bot \notin M'$. Since $M(\Delta) \subseteq M'$, M' also satisfies all of integrity constraints in I.

(2) Assume $M' = \min(T'^{M'})$ and $\bot \notin M'$. Let Δ be $M' \cap A$ and $M(\Delta)$ be $M' - \widetilde{V}$.

By Lemma 20, $\min(T'^{M'}) = \min((T \cup F(\Delta))^{M(\Delta)}) \cup \widetilde{V}$.

$$M' = M(\Delta) \cup \widetilde{V},$$

since $M(\Delta) = M' - \widetilde{V}$.

Therefore, by the assumption, $\min((T \cup F(\Delta))^{M(\Delta)}) \cup \widetilde{V} = M(\Delta) \cup \widetilde{V}$.

$$\min\left((T \cup F(\Delta))^{M(\Delta)} \cap \widetilde{V}\right) = M(\Delta) \cap \widetilde{V} = \emptyset \quad \text{then}$$
$$\min\left((T \cup F(\Delta))^{M(\Delta)}\right) = M(\Delta).$$

This means that $M(\Delta)$ is a stable model of $T \cup F(\Delta)$. Since $\bot \notin M'$ and $M(\Delta) \subseteq M'$, $\bot \notin M(\Delta)$.

Proof of Corollary 13. Suppose $M(\Delta)$ is a generalized stable model for $\langle T, A\rangle$ and $q \in M(\Delta)$. Therefore, $M(\Delta)$ is also a generalized stable model for $\langle T \cup \{\bot \leftarrow not\, q\}, A\rangle$. From Theorem 11, there exists a stable model M' for $T \cup \Gamma(A) \cup \{\bot \leftarrow not\, q\}$ such that $M' = M(\Delta) \cup \widetilde{V}$. Thus, $M' \cap A = \Delta$.

Suppose M' is a stable model for $T \cup \Gamma(A) \cup \{\bot \leftarrow not\, q\}$. Let Δ be $M' \cap A$ and $M(\Delta)$ be $M' - \widetilde{V}$. From Theorem 11, $M(\Delta)$ is a generalized stable model for $\langle T \cup \{\bot \leftarrow not\, q\}, A\rangle$. Therefore $q \in M(\Delta)$ (suppose $q \notin M(\Delta)$, then $\bot \in M(\Delta)$ by $\bot \leftarrow not\, q$ and the definition of stable model). □

Proof of Theorem 17.

1. We can show that the sequence of rules by *proc_O* (*proc_N*) can be selected along iterations of *proc_S* to output M.
2. Let T be a finite normal logic program with finite integrity constraints. Since the proof below is common to *proc_O* and *proc_N* before considering subprocedures *select_rule* and *topdown_select*, we do not explicitly mention *proc_N* until we have to distinguish between *select_rule* and *topdown_select*. Suppose *proc_S* outputs M with a sequence $R_0, \ldots, R_n$. We show by induction on the numbers of i of iterations of the main procedure in *proc_O* and k of iterations of *propagate* in *proc_O* that the following conditions hold:

   Condition 1. There are two sequences s_{modified} and s_{rest} such that

   1(a) *proc_S* outputs M with $s_{\text{modified}} \cdot s_{\text{rest}}$, where $\cdot$ is a concatenation of two sequences, and

   1(b) we can select rules along s_{modified} in *proc_O* up to i and k.

   Condition 2. $\widetilde{M}_i^k \cap M = \emptyset$

   Note that a set of propositions constructed in *proc_S* up to s_{modified} is equal to a set constructed in *proc_O* up to s_{modified} by Condition 1 and so, the set constructed in *proc_O* is a subset of M.

   If $i = 0$ and $k = 0$, let $s_{\text{modified}} = \langle\rangle$ (null sequence) and $s_{\text{rest}} = \langle R_0, \ldots, R_n\rangle$ (the original sequence).

   Then, Condition 1 clearly holds. And since $\widetilde{M}_i^k = \{\bot\}$, Condition 2 holds.

   Suppose up to i and k, the above conditions hold. We enter an iteration of *propagate*. We show the above conditions hold for i and $k+1$.

Condition 1(a). This condition should be checked if there is a rule which satisfies case 1 of *propagate*. Suppose there is such a rule R, that is, $pos(R) \subseteq M_i^k$ and $neg(R) \subseteq \widetilde{M}_i^k$. Then, $pos(R) \subseteq M$ and $(neg(R) \cap M) = \emptyset$ since M_i^k is equal to a set of propositions constructed by s_{modified} in *proc_S* and $(\widetilde{M}_i^k \cap M) = \emptyset$ by the inductive hypothesis. Therefore, R can be selected after this point by *proc_S* until $head\_of(R)$ is added. Since $head\_of(R)$ is included in M, there must be a rule R' in s_{rest} such that $head\_of(R') = head\_of(R)$. We delete R' from s_{rest} and add R to the tail of s_{modified} to obtain new s_{rest} and new s_{modified} for i and $k+1$.

Now, it is sufficient to show that *proc_S* outputs M with $s_{\text{modified}} \cdot s_{\text{rest}}$. Suppose *proc_S* does not output M with $s_{\text{modified}} \cdot s_{\text{rest}}$. There are two possibilities for this situation.

1. There is some R'' in s_{rest} which contains $head\_of(R)$ in $neg(R'')$.
2. There is some R'' in s_{rest} whose head is contained in $neg(R)$.

The first case is impossible since if such R'' is equal to R' or appears before R' in the previous s_{rest}, R' cannot be selected in the previous s_{rest}, and if such R'' appears after R' in the previous s_{rest}, R'' cannot be selected in the previous s_{rest}. The second case is also impossible since $(\widetilde{M}_i^k \cap M) = \emptyset$ by the inductive hypothesis and $head\_of(R'') \in M$ and $neg(R) \subseteq \widetilde{M}_i^k$.

Thus, Condition 1(a) holds for i and $k+1$.

Condition 1(b). As shown above, if there is a rule which satisfies case 1 of *propagate*, we can choose the rule along s_{modified} by *proc_O*.

However, we should also check if there is a rule satisfying case 3 since if there is such a rule, *proc_O* will fail.

Suppose there is a rule R satisfying case 3. Then, R must be eventually selected by *proc_S* since $(\widetilde{M}_i^k \cap M) = \emptyset$ by the inductive hypothesis or $head\_of(R)$ will be added by some other rule. In either case, $head\_of(R)$ must be included in M and this leads to contradiction because $head\_of(R) \in \widetilde{M}_i^k$, and $(\widetilde{M}_i^k \cap M) = \emptyset$. Therefore, there is no rule satisfying case 3.

Thus, Condition 1(b) holds for i and $k+1$.

Condition 2. This condition should be checked if there is a proposition added to $\widetilde{M}_i^{k+1}$ by case 2.

Suppose that there is a proposition p added to $\widetilde{M}_i^{k+1}$ by case 2, that is, $p \notin M_i^k$ and there is a rule R such that $head\_of(R) \in \widetilde{M}_i^k$ and $p \in pos(R)$ and $(pos(R) - \{p\}) \subseteq M_i^k$ and $neg(R) \subseteq \widetilde{M}_i^k$. Then, we show that M does not include p. Suppose M includes p. R must be eventually selected by *proc_S* since $(\widetilde{M}_i^k \cap M) = \emptyset$ by the inductive hypothesis or $head\_of(R)$ will be included by some other rule. In either case, $head\_of(R)$ must be included in M and this leads to contradiction because $head\_of(R) \in \widetilde{M}_i^k$, and $(\widetilde{M}_i^k \cap M) = \emptyset$. Therefore, M does not include p.

Thus, Condition 2 holds for i and $k+1$.

If this iteration (i and $k+1$) is not the last iteration in *propagate*, we can also prove that the above conditions hold for a new iteration (i and $k+2$). Otherwise, we return to the **until** loop and go to the **if** sentence. We cannot fail at the **if** sentence in the **until** loop since $M_i^{k+1} \subseteq M$ and $(\widetilde{M}_i^{k+1} \cap M) = \emptyset$. Therefore, we can go to *select_rule* (*topdown_select*) in the **until** loop.

Case of proc_O: In the case that $s_{\text{rest}} = \langle\rangle$, we assume there exists a rule satisfying the **if** condition in *select_rule*. This contradicts that *proc_S* outputs an answer with

$s_{\text{modified}} \cdot s_{\text{rest}}$. Since *select_rule* returns *nil_rule* in this case, *proc_O* outputs M with s_{modified}.

We show that *select_rule* returns some rule, if $s_{\text{rest}} \neq \langle\rangle$. In this case, *select_rule* can return the left-most rule R ($head\_of(R) \notin \widetilde{M}_i^k$ because $head\_of(R) \in M$) in s_{rest}. We then delete the left-most rule in s_{rest} and add the rule to the tail of s_{modified}. This means that we can confirm Conditions 1 and 2 hold for $i+1$ and 0 after *select_rule* in *proc_O*. As a result, the induction for *proc_O* is proved.

Case of proc_N: In the case that $s_{\text{rest}} = \langle\rangle$ and the case that $s_{\text{rest}} \neq \langle\rangle$ such that there is no rule satisfying the **if** conditions in *topdown_select*, we prove the induction as same as the case of *proc_O*.

Next we consider the other case that there is a rule (or some rules) satisfying the **if** conditions in *topdown_select*. At first we show that *topdown_check* returns some rule R from s_{rest}, which is not necessarily the left-most rule in s_{rest}. Since, in the next lemma, we see that the *topdown_check* returns a rule in s_{rest}, we can confirm that Conditions 1 and 2 hold for $i+1$ and 0 after we delete the returned rule R by *topdown_check* from s_{rest} and add R to the tail of s_{modified}.

Lemma 21. *topdown_check returns a rule in* s_{rest}.

Proof. We show the following propositions to show the lemma. Based on the propositions *topdown_check* eventually terminates (without failure) and returns the rule selected at the last iteration. That is because $th(R_j) > 0$ for all j and $th(R_j) > th(R_{j+1})$. ($th(R) = n$ means R is the nth rule in s_{rest}, and j is the number of iterations of the **goto** loop in *topdown_check*.)

1. *topdown_check* does not fail and can select R_j from s_{rest} at **select**.
2. If R_j and R_{j+1} are in s_{rest}, $th(R_j) > th(R_{j+1})$.

1. We show by induction on j. If $j = 0$, that means $Pos = \{p\}$, there must be a rule in s_{rest} whose head is p (the case where there is not such a rule contradicts the fact that *proc_S* outputs the answer with $s_{\text{modified}} \cdot s_{\text{rest}}$).

Suppose up to j, *topdown_check* does not fail. If $pos(R_j) - M \neq \emptyset$, every proposition $pr \in pos(R_j) - M$ should have a rule R_{pr} in s_{rest} which satisfies $head\_of(R_{pr}) = pr$ and the other three conditions at the **if** sentence (otherwise R_j cannot be selected in s_{rest} by *proc_S*). So we can select the rule from s_{rest} at **select** in *topdown_check*.

2. By 1 all R_j can be in s_{rest}. Suppose $th(R_j) = th(R_{j+1})$, which means $R_j = R_{j+1}$. Then, for some $p \in pos(R_j) - M$, $head\_of(R_{j+1}) = p$. This contradicts $head\_of(R_j) \notin pos(R_j)$.

Suppose $th(R_j) < th(R_{j+1})$. There exists $p \in pos(R_j) - M$ s.t. $head\_of(R_{j+1}) = p$. But this contradicts that *proc_S* outputs the answer with $s_{\text{modified}} \cdot s_{\text{rest}}$.

References

[1] A. Bondarenko, F. Toni, R.A. Kowalski, An assumption-based framework for non-monotonic reasoning, LPNMR'93 (1993) 171–189.

[2] A. Bondarenko, P.M. Dung, R.A. Kowalski, F. Toni, An abstract, argumentation-theoretic approach to default reasoning, Artificial Intelligence 93 (1–2) (1997) 63–101.

[3] W. Chen, D.S. Warren, Tabled evaluation with delaying for general logic programs, Journal of the ACM 43 (1) (1996) 20–74.
[4] P. Cholewiński, V.W. Marek, M. Truszczyński, Default reasoning system DeReS, in: Proceedings of KR'96, 1996, pp. 518–528.
[5] P. Cholewiński, Reasoning with stratified default theories, LPNMR'95 (1995) 273–286.
[6] M. Denecker, D. De Schreye, SLDNFA: an abductive procedure for normal abductive programs, in: Proceedings of JICSLP'92, 1992, pp. 686–700.
[7] M. Denecker, D. De Schreye, SLDNFA: an abductive procedure for abductive logic programs, Journal of Logic Programming 32 (2) (1998) 111–167.
[8] J. Doyle, A truth maintenance system, Artificial Intelligence 12 (1979) 231–272.
[9] P.M. Dung, Negations as hypotheses: an abductive foundation for logic programming, in: Proceedings of ICLP'91, 1991, pp. 3–17.
[10] C. Elkan, A rational reconstruction of nonmonotonic truth maintenance systems, Artificial Intelligence 43 (1990) 219–234.
[11] K. Eshghi, R.A. Kowalski, Abduction compared with negation by failure, in: Proceedings of ICLP'89, 1989, pp. 234–254.
[12] K. Eshghi, Computing stable models by using the ATMS, in: Proceedings of AAAI'90, 1990, pp. 272–277.
[13] F. Fages, A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics, in: Proceedings of ICLP'90, 1990, pp. 442–458.
[14] Y. Fujiwara, S. Honiden, Relating the TMS to autoepistemic logic, in: Proceedings of IJCAI'89, 1989, pp. 1199–1205.
[15] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of LP'88, 1988, pp. 1070–1080.
[16] L. Giordano, A. Martelli, Generalized stable models truth maintenance and conflict resolution, in: Proceedings of ICLP'90, 1990, pp. 427–441.
[17] K. Inoue, M. Koshimura, R. Hasegawa, Embedding negation as failure into a model generation theorem prover, in: Proceedings of the 11th International Conference Automated Deduction, Lecture Notes in Artificial Intelligence, vol. 697, 1992, pp. 400–415.
[18] K. Inoue, C. Sakama, Transforming abductive logic programs to disjunctive programs, in: Proceedings of ICLP'93, 1993.
[19] U. Junker, K. Konolige, Computing the extensions of autoepistemic and default logics with a truth maintenance system, in: Proceedings of AAAI'90, 1990, pp. 278–223.
[20] A.C. Kakas, P. Mancarella, On the relation between truth maintenance and abduction, in: Proceedings of PRICAI'90, 1990, pp. 438–443.
[21] A.C. Kakas, P. Mancarella, Generalized stable models: a semantics for abduction, in: Proceedings of ECAI'90, 1990, pp. 385–391.
[22] A.C. Kakas, R.A. Kowalski, R. Toni, Abductive logic programming, Journal of Logic and Computation 2 (6) (1992) 719–770.
[23] R. Manthey, F. Bry, SATCHMO: a theorem prover in PROLOG, CADE-88, 1988.
[24] I. Niemelä, P. Simons, Efficient implementation of the well-founded and stable model semantics, in: Proceedings of JICSLP'96, 1992, pp. 671–685.
[25] I. Niemelä, P. Simons, Smodels – an implementation of the stable model and well-founded semantics for normal logic programs, in: Proceedings of LPNMR'97, 1997.
[26] S.G. Pimentel, J.L. Cuadrado, A truth maintenance system based on stable models, in: Proceedings of NACLP'89, 1989, pp. 274–290.
[27] R. Ramakrishnan, Magic templates: a spellbinding approach to logic programming, in: Proceedings of ICLP'88, 1988.
[28] A. Ramsay, Generating relevant models, Journal of Automated Reasoning 7 (1991) 359–368.
[29] P. Rao, K. Sagonoas, T. Swift, D. Warren, J.S. Freire, XSB: a system for efficiently computing well-founded semantics, LPNMR'97, 1997.
[30] M. Reinfrank, O. Dressler, G. Brewka, On the relation between truth maintenance and autoepistemic logic, in: Proceedings of IJCAI'89, 1989, pp. 1206–1212.
[31] D. Saccà, C. Zaniolo, Stable models and non-determinism in logic programs with negation, in: Proceedings of PODS'90, 1990, pp. 205–217.
[32] K. Satoh, N. Iwayama, E. Sugino, K. Konolige, An implementation of TMS in concurrent logic programming language, Preliminary Report, ICOT-TR-568, ICOT 1990.

[33] K. Satoh, N. Iwayama, Computing abduction by using TMS, in: Proceedings of ICLP'91, 1991, pp. 505–518.
[34] K. Satoh, N. Iwayama, A query evaluation method for abductive logic programming, in: Proceedings of JICSLP'92, 1992, pp. 671–685.
[35] F. Toni, R.A. Kowalski, Reduction of abductive logic programs to normal logic programs, in: Proceedings of ICLP'95, 1995, pp. 367 –381.