NORTH-HOLLAND

THE IFF PROOF PROCEDURE FOR ABDUCTIVE LOGIC PROGRAMMING

TZE HO FUNG AND ROBERT KOWALSKI

▷ In this paper, we outline a proof procedure which combines reasoning with defined predicates together with reasoning with undefined, *abducible*, predicates. Defined predicates are defined in if-and-only-if form. Abducible predicates are constrained by means of integrity constraints. Given an initial query, the task of the proof procedure is to construct a definition of the abducible predicates and a substitution for the variables in the query, such that both the resulting instance of the query and the integrity constraints are implied by the extended set of definitions.

The iff proof procedure can be regarded as a hybrid of the proof procedure of Console et al. and the SLDNFA procedure of Denecker and De Schreye. It consists of a number of inference rules which, starting from the initial query, rewrite a formula into an equivalent formula. These rules are: 1) *unfolding*, which replaces an atom by its definition; 2) *propagation*, which resolves an atom with an implication; 3) *splitting*, which uses distributivity to represent a goal as a disjunction of conjunctions; 4) *case analysis* for an equality $X = t$ in the conditions of an implication, which considers the two cases $X = t$ and $X \neq t$; 5) *factoring* of two abducible atoms, which considers the two cases, where the atoms are identical and where they are different, 6) *rewrite rules for equality*, which simulate the unification algorithm; and 7) *logical simplifications*, such as $A \wedge false \leftrightarrow false$.

The proof procedure is both sound and complete relative to the three-valued completion semantics. These soundness and completeness results improve previous results obtained for other proof procedures. ◁

Address correspondence to Robert Kowalski, Department of Computing, Imperial College, London SW7 2B2, England.

Received July 1996; accepted December 1996.

1. INTRODUCTION

The iff proof procedure can be regarded as a hybrid of the proof procedure of Console et al. [1] and the SLDNFA procedure of Denecker and De Schreye [2]. All three proof procedure use the if-and-only-if form of a logic program to specify its semantics.

Like the Console et al. proof procedure, but unlike SLDNFA, the iff proof procedure uses unfolding with if-and-only-if definitions to replace a formula by an equivalent formula, starting with an initial query. However, the iff proof procedure augments the unfolding rule with other inference rules, resembling those employed by SLDNFA, to obtain stronger completeness results than those obtained with the other proof procedures. The iff proof procedure is simpler than SLDNFA, largely because of its use of iff definitions, its use of explicit rewrite rules for equality, and its use of existential quantifiers and free variables to avoid skolemization.

We have investigated a number of applications of the proof procedure and some of its variants. These applications include constraint logic programming [10, 11, 14], planning [6], semantic query optimization [11], and deductive database view updates [4–6]. For this last application, we have developed a modified form of the iff proof procedure, which minimizes changes to an existing database of abducibles.

We have also used the proof procedure to implement the reasoning component of an agent that combines deliberative, goal-oriented reasoning with reactive, condition–action rule behavior [15, 16]. Goal-oriented reasoning is obtained by using unfolding to reduce atomic goals to alternative conjunctions of subgoals. Condition–action rule behavior is obtained by using propagation to trigger integrity constraints by means of updates.

In the remainder of this paper, we define our syntax and semantics for abductive logic programs (Section 2), define the proof procedure (Section 3), explain how answers are extracted from derivations (Section 4), state soundness and completeness results (Section 5), and conclude (Section 6).

Detailed proofs of the results in Section 5 can be found in [6].

2. ABDUCTIVE LOGIC PROGRAMS AND THEIR SEMANTICS

We define an abductive logic program to be a tuple $\langle T, IC, Ab \rangle$ where

1. T is a set of *definitions* in iff form:

$$P(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n \qquad k, n \geq 0$$

where P is a *defined* predicate symbol different from $=$, *true*, *false*, and any predicate symbol in Ab, the variables $X_1, \ldots, X_k$ are all distinct, and each D_i is a conjunction of literals. A *literal* is an atom or the negation of an atom. Negative literals $\neg A$ are rewritten as implications in the form $false \leftarrow A$. Every defined predicate has only one definition. For every variable X_i and disjunct D_j, the disjunct contains exactly one literal of the form $X_i = t$, for some term t.[1]

[1] This last condition formalizes the requirement that definitions in iff form are the completions of logic programs in if form.

The variables $X_1, \ldots, X_k$ are implicitly universally quantified, with the scope being the entire definition. Any variable in a disjunct D_i of a definition which is not one of $X_1, \ldots, X_k$ is implicitly existentially quantified, with the scope being the disjunct. The atom $P(X_1, \ldots, X_k)$, defined in a definition in T, is said to be the *head* of the definition. The disjunction $D_1 \vee \ldots \vee D_n$ is its *body*. When $n = 0$, the disjunction is equivalent to *false*.

2. IC (the set of *integrity constraints*) is a consistent finite set (or, equivalently, a conjunction) of *implications*:

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n \qquad m, n \geq 0$$

where each A_i and B_j is an atom.

All variables in an integrity constraint are implicitly universally quantified, with the scope being the entire implication. The disjunction $A_1 \vee \ldots \vee A_m$ in the implication is the *conclusion* of the implication, and the conjunction $B_1 \wedge \ldots \wedge B_n$ is its *condition*. When $m = 0$, the disjunction is equivalent to *false*. When $n = 0$, the conjunction is equivalent to *true*.

3. Ab is the set of all predicate symbols, called *abducible*, different from $=$, *true*, *false*, and from any predicate symbol defined in T.

A *query* is a formula of the form

$$B_1 \wedge \ldots \wedge B_n \qquad n \geq 1$$

where each B_i is a literal. All variables in a query are free.

To simplify the treatment of quantifiers, so that they can be implicit rather than explicit, we require that definitions, integrity constraints, and queries all be *allowed*. A definition $P(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n$ is *allowed* if (and only if) every variable other than $X_1, \ldots, X_k$ occurring in any D_i occurs in a positive nonequality atom in D_i. Similarly, a query is *allowed* if (and only if) any variable occurring in the query occurs in a positive atom in the query. An integrity constraint is *allowed* if (and only if) every variable in the conclusion occurs in the condition.

As in SLDNF, the allowedness restrictions ensure that queries never flounder. In practice, much weaker conditions are adequate. In particular, as in SLDNF and in SLDNFA, an appropriate safety condition can be employed instead.

The *semantics* of abductive logic programs is given by specifying what constitutes an answer to a query. Intuitively, an answer is a set of definitions for the abducible predicates which, together with the initially given set of definitions T, implies an instance of the query and satisfies the integrity constraints. For simplicity, we restrict definitions of abducibles to completions of *ground* (i.e., variable-free) atoms.

More precisely, given an abductive logic program $\langle T, IC, Ab \rangle$, an *answer* to a query Q is a pair (D, σ) where D is a finite set of ground abducible atoms, and σ is a substitution of ground terms for the variables in Q, such that

1. $T \cup \mathrm{Comp}(D) \cup CET \models Q\sigma$ and
2. $T \cup \mathrm{Comp}(D) \cup CET \models IC$.

Here, $\mathrm{Comp}(D)$ is the *completion* of D, consisting of iff definitions of the predicate symbols occurring in D. Abducible predicates which do not occur in D are defined as *false* in $\mathrm{Comp}(D)$. CET is the Clark equality theory. $\models$ is the logical consequence relation for three-valued logic, as defined in [12]. The ground terms in the substitution σ come from some language which includes, but need not be identical to, the vocabulary of $T \cup Q \cup IC$.

Note that condition 2) states the *theoremhood* view of integrity constraint satisfaction. In contrast, the *consistency* view requires that $T \cup \text{Comp}(D) \cup CET \cup IC$ be consistent. Because theoremhood is semi-decidable and consistency is not, it is possible to develop a complete proof procedure for the theoremhood view, but not for the consistency view. However, any answer according to the theoremhood view is also an answer according to the consistency view because $T \cup \text{Comp}(D) \cup CET$ is always consistent in three-valued logic.

It is often desirable to give preference to *minimal answers* (D, σ) where there is no answer (D', σ), where D' is strictly contained in D.

Example 2.1.

T: grass-is-wet $\leftrightarrow$ rain-last-night $\vee$ sprinker-was-on
IC: cloudy-last-night $\leftarrow$ rain-last-night
false $\leftarrow$ cloudy-last-night
Ab: rain-last-night, sprinker-was-on, cloudy-last-night
Q: grass-is-wet

The query has only one answer, which is also minimal, ({sprinkler-was-on}, $\emptyset$), where $\emptyset$ is the empty substitution.

Example 2.2 (Based on [3]).

T: lamp(X) $\leftrightarrow$ $X = a$
battery(X,Y) $\leftrightarrow$ $X = b \wedge Y = c$
faulty-lamp $\leftrightarrow$ [lamp(X) $\wedge$ broken(X)] $\vee$
[power-failure(X) $\wedge$ $\neg$backup(X)]
backup(X) $\leftrightarrow$ battery(X, Y) $\wedge$ $\neg$empty(Y)
IC: $\emptyset$ (i.e., the empty set)
Ab: broken, power-failure, empty
Q: faulty-lamp

The query has minimal answers:

({broken(a)}, $\emptyset$)
({power-failure(b), empty(c)}, $\emptyset$)
({power-failure(t)}, $\emptyset$), where t is any ground term of the language and $t \neq b$.

It also has many more nonminimal answers.

3. THE IFF PROOF PROCEDURE

The iff proof procedure is a rewriting procedure, consisting of a number of inference rules, each of which replaces a formula by one which replaces a formula by one which is equivalent to it in the theory $T \cup CET$. A *derivation* of a formula F_n, starting from a formula F_1, is a sequence of formulas $F_1, \ldots, F_n$ such that each F_{i+1} in the sequence is obtained from the previous formula F_i by application of

one of the inference rules. Therefore, any such derivation has the property that

$T \cup CET \models F_1 \leftrightarrow F_n$.

Every formula F_i in a derivation is a disjunction:

$D_1 \vee \ldots \vee D_n \qquad n \geq 0.$

If $n = 0$, the disjunction is equivalent to *false*. Each disjunct D_i is a conjunction

$C_1 \wedge \ldots \wedge C_m \qquad m \geq 0.$

If $m = 0$, the conjunction is equivalent to *true*. Each conjunct C_i is either

an atom,

a disjunction of conjunctions of literals, or

an implication (as defined earlier).

Given an initial query Q, the proof procedure constructs an *initial formula* F_1, which is Q conjoined with the set of integrity constraints. F_1, therefore, is a degenerate disjunction, with a single disjunct, which is a conjunction whose conjuncts are

the positive atoms in Q,

the negative literals $\neg A$ in Q, written as implications *false* $\leftarrow A$, and

the implications in IC.

Any variables in IC are renamed so that they are distinct from the variables in Q.

As a result of the allowedness restrictions, the inference rules preserve the unambiguous reading of the implicit quantification of variables in the formulas F_i of a derivation. Every free variable of the initial query occurs free in each disjunct of F_i. Any other variable in an atom occurring directly as a conjunct is implicitly existentially quantified, with the scope being the disjunct in which the atom occurs. All remaining variables occur in implications and are implicitly universally quantified, with the scope being the implication itself. Note that, except for the integrity constraints, implications in a derivation may contain free or existentially quantified variables.

The construction of a derivation has a "procedural" interpretation in terms of generating an or-tree in search of answers to the initial query. Formulas in the derivation correspond to successive *frontiers* of the search tree. The disjuncts D_i in a frontier correspond to *nodes* of the search tree. Conjuncts in a node are *goals* to be satisfied.

Each step in a derivation is the application of an inference rule which replaces a nonleaf node in a frontier by one or two successor nodes. Different nodes and different goals within a node may be *selected* for the application of an inference rule. For example, systematically selecting the leftmost node D_1 in a frontier gives rise to depth-first search. Other strategies for selecting nodes give rise to other search strategies. The strategy for selecting a goal within a node is analogous to the selection rule of SLDNF. As in SLDNF, any goal may be selected, provided the selection strategy is "fair."

A *leaf node* is a node which has no successor nodes in the sense that no new inference can be applied to the node. A node containing a conjunct *false* is also a *leaf node*, called a *failure node*. A failure node is itself equivalent to *false*, and no inference rules can be applied to it.

Answers (D, σ) to a query Q are extracted from nonfailure leaf nodes D_i. First, a ground substitution σ' is constructed which satisfies the equalities and disequalities[2] in D_i and arbitrarily instantiates any remaining variables in D_i which are not universally quantified. D is obtained by applying σ' to the abducible atoms which are conjuncts of D_i. σ is the restriction of σ' to the variables occurring in Q. Answer extraction is explained in greater detail in Section 4, and is illustrated in Examples 3.1 and 3.2 below.

The *inference rules* of the iff proof procedure consist of:

1. *unfolding*, which creates a single successor node by replacing an atom (which occurs directly as a conjunct in a node or in the condition of an implication) by the body of its definition in T,
2. *propagation*, which creates a single successor node by resolving an atom in a node with an implication in the same node,
3. *splitting*, which creates two successor nodes, using the law of distributivity.
4. *case analysis* for an equality $X = t$ in the condition of an implication, which creates two successor nodes, one for the case $X = t$, and one for the case $X \neq t$.
5. *factoring* of two abducible atoms, which creates two successor nodes, one for the case where the two atoms are identical, and one for the case where they are different,
6. *rewrite rules for equality*, which simulate the unification algorithm,
7. *logical simplifications*, which replace the left-hand side of a logical equivalence by the right-hand side, such as

$$A \wedge false \leftrightarrow false$$

$$A \vee false \leftrightarrow A$$

$$A \wedge true \leftrightarrow A$$

$$A \vee true \leftrightarrow true$$

$$[A \leftarrow true] \leftrightarrow A$$

$$[A \leftarrow false] \leftrightarrow true$$

$$\neg A \leftrightarrow [false \leftarrow A]$$

$$[B \leftarrow C \wedge \neg A] \leftrightarrow [A \vee B \leftarrow C].$$

For simplicity of exposition, we regard the rewrite rules for equality and logical simplifications as simplifying the form of a node, rather than as creating a new node.

Before defining the proof procedure formally, we illustrate it by means of the two examples introduced in the previous section.

[2] The *disequality* $s \neq t$ is used as an abreviation for $false \leftarrow s = t$.

Example 3.1.

T: grass-is-wet $\leftrightarrow$ rain-last-night $\vee$ sprinkler-was-on
IC: cloudy-last-night $\leftarrow$ rain-last-night
false $\leftarrow$ cloudy-last-night
Ab: rain-last-night, sprinkler-was-on, cloudy-last-night
Q: grass-is-wet
F_1 grass-is-wet $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]
F_2 [rain-last-night $\vee$ sprinkler-was-on] $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]
F_3 [rain-last-night $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]] $\vee$
[sprinker-was-on $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]]
F_4 [rain-last-night $\wedge$ cloudy-last-night $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]] $\vee$
[sprinkler-was-on $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last night]]
F_5 [rain-last-night $\wedge$ cloudy-last-night $\wedge$ *false* $\wedge$
[cloudy-last-night $\leftarrow$ rain-last-night] $\wedge$ [*false* $\leftarrow$ cloudy-last-night]] $\vee$
[sprinkler-was-on $\wedge$ [cloudy-last-night $\leftarrow$ rain-last-night]
$\wedge$ [*false* $\leftarrow$ cloudy-last-night]]

F_2 is obtained by unfolding, F_3 by splitting, and both F_4 and F_5 by propagation. All nodes in F_5 are leaf nodes. An answer can be extracted from the only nonfailure leaf node of F_5. That answer coincides with the unique answer ({sprinkler-was-on}, $\emptyset$) given by the semantics.

Example 3.2.

T: lamp(X) $\leftrightarrow$ $X = a$
battery(X, Y) $\leftrightarrow$ $X = b \wedge Y = c$
faulty-lamp $\leftrightarrow$ [lamp(X) $\wedge$ broken(X)] $\vee$
[power-failure(X) $\wedge$ $\neg$backup(X)]
backup(X) $\leftrightarrow$ battery(X, Y) $\wedge$ $\neg$empty(Y)
IC: $\emptyset$
Ab: broken, power-failure, empty
Q: faulty-lamp
F_1 faulty-lamp
F_2 [lamp(X) $\wedge$ broken(X)] $\vee$ [power-failure(X) $\wedge$ $\neg$backup(X)]
$\leftrightarrow$ [lamp(X) $\wedge$ broken(X)] $\vee$ [power-failure(X) $\wedge$ [false $\leftarrow$ backup(X)]]
F_3 [$X = a$ $\wedge$ broken(X)] $\vee$ [power-failure(X) $\wedge$ [false $\leftarrow$ backup(X)]]
$\leftrightarrow$ [$X = a$ $\wedge$ broken(a)] $\vee$ [power-failure(X) $\wedge$ [false $\leftarrow$ backup(X)]]
F_4 [$X = a$ $\wedge$ broken(a)] $\vee$
[power-failure(X) $\wedge$ [false $\leftarrow$ [battery(X, Y) $\wedge$ $\neg$empty(Y)]]]
$\leftrightarrow$ [$X = a \wedge$ broken(a)] $\vee$
[power-failure(X) $\wedge$ [empty(Y) $\leftarrow$ battery(X, Y)]]
F_5 [$X = a \wedge$ broken(a)] $\vee$
[power-failure(X) $\wedge$ [empty(Y) $\leftarrow$ $X = b \wedge Y = c$]]

$\leftrightarrow [X = a \wedge \text{broken}(a)] \vee$
$\quad [\text{power-failure}(X) \wedge [\text{empty}(c) \leftarrow X = b]]$
F_6 $[X = a \wedge \text{broken}(a)] \vee$
$\quad [\text{power-failure}(X) \wedge [X \neq b \vee [X = b \wedge \text{empty}(c)]]]$
F_7 $[X = a \wedge \text{broken}(a)] \vee$
$\quad [\text{power-failure}(X) \wedge X \neq b] \vee [\text{power-failure}(X) \wedge X = b \wedge \text{empty}(c)]$
$\leftrightarrow$ $[X = a \wedge \text{broken}(a)] \vee$
$\quad [\text{power-failure}(X) \wedge X \neq b] \vee [\text{power-failure}(b) \wedge X = b \wedge \text{empty}(c)]$

Here, F_2, F_3, F_4, and F_5 are all obtained by unfolding. F_6 is obtained by case analysis, and F_7 is obtained by splitting. The equivalences in F_2 and F_4 rewrite negative literals as atoms in implications. The equivalences in F_3, F_5, and F_7 apply rewrite rules for equality. In F_3, the rewrite rule applies the substitution X/a only to the disjunct containing $X = a$. This respects the fact that X is existentially quantified. No new application of the inference rules can be performed on F_7. Answers can be extracted from each disjunct of F_7. These answers coincide with the minimal answers specified by the semantics.

We now define the inference rules more formally.

3.1. *Unfolding*

Given an atom $P(t_1, \ldots, t_k)$ (occurring either directly as a conjunct in a node or in the condition of an implication and a definition

$$P(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n$$

unfolding replaces the atom by

$$D_1\theta \vee \ldots \vee D_n\theta \qquad \text{where } \theta = \{X_1/t_1, \ldots, X_k/t_k\}$$

leaving the rest of the node unchanged.

Variables introduced into a node by unfolding are "standardized apart," so they do not accidentally become identical to other variables in the node.

When unfolding is applied to an atom $P(t_1, \ldots, t_k)$ in the condition of an implication

$$A \leftarrow P(t_1, \ldots, t_k) \wedge B$$

the resulting implication is rewritten, equivalently, as a conjunction of implications:

$$[A \leftarrow D_1\theta \wedge B] \wedge \ldots \wedge [A \leftarrow D_n\theta \wedge B].$$

For simplicity, we have written the atom $P(t_1, \ldots, t_k)$ as the first atom in the condition of the implication. Any other atom in the condition can be selected for unfolding. Any negative literals in $D_i\theta$ are rewritten as positive atoms in the conclusion of the implication.

Unfolding is not applied to atoms in conclusions of implications. Such unfolding, in effect, is deferred until the condition of the implication has been eliminated, and therefore until any universally quantified variables in the atom have been eliminated (which is always possible because of the allowedness restriction).

The following example shows that unfolding respects the three-valued, rather than the two-valued semantics.

T: $p \leftrightarrow \neg p$
Q: p
F_1 p
F_2 $\textit{false} \leftarrow p$
F_3 p
.
.
.

The proof procedure loops without termination, reflecting the fact that T has a three-valued model in which p is *undefined*.

In the two-valued semantics, T is inconsistent, and therefore implies both p and $\neg p$. Therefore, the proof procedure is incomplete for the two-valued semantics, which is consistent with Theorems 2 and 3 in Section 5.

3.2. *Propagation*

Given two conjuncts in the same node, one of which is an atom $P(s_1, \ldots, s_k)$, and the other of which is an implication $A \leftarrow P(t_1, \ldots, t_k) \wedge B$, *propagation* generates a single successor node by adding the implication

$$A \leftarrow t_1 = s_1 \wedge \ldots \wedge t_k = s_k \wedge B$$

to the node. The equalities are handled by the rewrite rules and by case analysis.

The predicate symbol P, used for propagation, can be any predicate symbol, except equality. It is possible to restrict P to be abducible, without loss of completeness. However, the case where propagation is used with a defined predicate can improve efficiency, as the following example shows:

T: $p \leftrightarrow p$
IC: $\textit{false} \leftarrow p$
Q: p

With propagation, the proof procedure terminates by generating *false*. Without propagation, the proof procedure goes into an infinite loop.

3.3. *Splitting*

Given a node of the form

$$[C \vee D] \wedge C'$$

(where C and C' are conjunctions and D is a disjunction), *splitting* generates two successor nodes

$$[C \wedge C'] \vee [D \wedge C'].$$

Without splitting, the search tree corresponding to a formula in a derivation would be an arbitrary and–or tree (ignoring variables). Systematic use of splitting, with higher priority than the other inference rules, flattens the and–or tree so that it is an or-tree, each of whose nodes is a conjunction of atoms and implications.

Splitting also can be applied to the conclusion of an implication in the node:

$$[[A \vee D] \leftarrow B] \wedge C$$

when one of the disjuncts A is an atom which contains no universally quantified variables (but possibly contains free or existentially quantified variables). *Splitting* rewrites the node as two nodes

$$[A \wedge C] \vee [[D \leftarrow B] \wedge C].$$

Splitting also applies when the conclusion is a single atom A, in which case D is trivially equivalent to *false*.

Splitting conclusions of implications is necessary for completeness, as the following example shows:

T: $p \leftrightarrow p$
$r \leftrightarrow p \wedge \neg q$
Ab: q
Q: $\neg r$
F_1 $false \leftarrow r$ unfold r:
F_2 $q \leftarrow p$ split q:
F_3 $q \vee [false \leftarrow p]$

Although the proof procedure loops forever, repeatedly unfolding the atom p in the second node of F_3, an answer $(\{q\}, \emptyset)$ can be extracted from the first node of F_3.

The completeness theorem 2 of Section 5 improves previous completeness theorems obtained for related proof procedures [1, 2] because it applies in such cases. Previous completeness theorems hold only for the case where the proof procedure terminates, as in the case of our completeness theorem 3 in Section 5 below.

3.4. *Case Analysis (for an Equality)*

Given a node of the form

$$[A \leftarrow X = t \wedge B] \wedge C$$

where X is free or existentially quantified, t does not contain X, and t is not a universally quantified variable,[3] *case analysis* gives rise to two successor nodes:

$$[X = t \wedge [A \leftarrow B] \wedge C] \vee [X \neq t \wedge C].$$

The first disjunct corresponds to the case $X = t$. Any variables in t which are universally quantified in the implication correctly become existentially quantified in the first disjunct. The second disjunct corresponds to the case $X \neq t$. Any variables in t which are universally quantified in the implication correctly remain universally quantified in the second disjunct.

In theory, case analysis also could be applied to abducible atoms. However, this would give rise to nonminimal answers. By not applying case analysis to abducible atoms, the proof procedure and the answer extraction process effectively assume

[3]The equality rewrite rules deal with all other cases.

such atoms are false by default, unless they occur directly as conjuncts in the same node for some other reason. Case analysis is not applied to defined predicates because the truth value of such predicates may be *undefined* in the three-valued semantics.

3.5. *Factoring*

Given a node of the form

$$P(t_1, \ldots, t_k) \wedge P(s_1, \ldots, s_k) \wedge C$$

where P is abducible, *factoring* gives rise to two new nodes:

$$[P(t_1, \ldots, t_k) \wedge P(s_1, \ldots, s_k) \wedge [false \leftarrow t_1 = s_1 \wedge \ldots \wedge t_k = s_k] \wedge C]$$
$$\vee [P(t_1, \ldots, t_k) \wedge t_1 = s_1 \wedge \ldots \wedge t_k = s_k \wedge C].$$

Factoring is related to answer extraction. It separates answers in which abducible atoms are merged from answers in which they are distinct.

3.6. *Rewrite Rules for Equality*

These rules are adapted from Martelli and Montanari [13], and are applied both to an equality which occurs directly as a conjunct in a node and to an equality which occurs in the condition of an implication. They are not applied to an equality in the conclusion of an implication because (as with unfolding) such rewrites can be deferred until the condition of the implication has been eliminated.

1. Replace $f(t_1, \ldots, t_k) = f(s_1, \ldots, s_k)$ by $t_1 = s_1 \wedge \ldots \wedge t_k = s_k$.
2. Replace $f(t_1, \ldots, t_k) = g(s_1, \ldots, s_l)$ by *false* whenever f and g are distinct, $k, l \geq 0$.
3. Replace $t = t$ by *true* for any term t.
4. Replace $X = t$ by *false* whenever t is a term containing X.

5a. Replace $t = X$ by $X = t$ whenever X is a variable and t is not.
b. Replace $Y = X$ by $X = Y$ whenever X is a universally quantified variable and Y is not.

6a. If $X = t$ occurs as a conjunct in a node and X does not occur in t, then apply the substitution X/t to the entire node, retaining the conjunct $X = t$ intact.
6b. If $X = t$ occurs in the condition of an implication, X does not occur in t, and X is universally quantified, then apply the substitution X/t to the implication, deleting the equality.

Note that case analysis deals with the case where $X = t$ occurs in the condition of an implication, and neither X nor t are universally quantified variables.

3.7. *Logical Equivalences*

These equivalences, given earlier in this section, simplify formulas and put them in a form which facilitates application of the other inference rules.

4. ANSWER EXTRACTION

Answers (D, σ) can be extracted from a nonfailure leaf node N by first constructing a substitution σ' such that:

σ' replaces all variables in N which are not universally quantified by ground terms, and

σ' satisfies the equalities and disequalities in N.

The fact that the rewrite rules for equality have been exhaustively applied to the node ensures that one or more such substitutions σ' exist. (In the general case, it is necessary to assume that the language contains an infinite number of function symbols.) The substitution σ is the restriction of σ' to the variables occurring in Q. The set D is the set of all abducible atoms that are conjuncts in $N\sigma'$.

Notice that, by construction, $Q\sigma' = Q\sigma$ and $IC\sigma' = IC$. Moreover, CET implies all equalities and disqualities in $N\sigma'$, and D implies all abducible atoms that are conjuncts in $N\sigma'$.

Because N is a leaf node, propagation has been exhaustively applied with the atoms used to construct D. As a result, it is possible to show that Comp(D) implies all implications in $N\sigma'$ which are not disequalities. Therefore,

1. $\text{Comp}(D) \cup CET \models N\sigma'$.

It is easy to see that $T \cup CET \models F_1 \leftrightarrow F_n$ for any formula F_n in a derivation. Therefore,

2. $T \cup CET \models [Q \wedge IC] \leftarrow N$ and
3. $T \cup CET \models [Q \wedge IC]\sigma' \leftarrow N\sigma'$

which shows that (D, σ) is an answer to the query Q because, by 1) and 3),

$T \cup \text{Comp}(D) \cup CET \models Q\sigma$ and

$T \cup \text{Comp}(D) \cup CET \models IC$.

Moreover, by construction, D is the smallest set of abducible ground atoms such that $\text{Comp}(D) \cup CET \models N\sigma'$ where σ' is the auxiliary substitution used to construct σ. Thus, the answer (D, σ) is minimal in this sense.

5. SOUNDNESS AND COMPLETENESS

More formally, we can prove the following soundness and completeness theorems:

Theorem 1 (Soundness). *Let Q be a query to an abductive logic program $\langle T, IC, Ab \rangle$.*

1. Let (D, σ) be extracted from a nonfailure leaf node N in a derivation from $Q \wedge IC$. Then (D, σ) is an answer to the query Q.
   Moreover, if (D', σ) is any answer such that $\text{Comp}(D') \models N\sigma'$ where σ' is the auxiliary substitution used to construct σ, then D is a subset of D'.
2. If there exists a derivation from $Q \wedge IC$ of a formula
   $D_1 \vee \ldots \vee D_m$
   where each disjunct D_i is a failure node, then
   $T \cup CET \cup IC \models \neg Q$.

Theorem 2 (Completeness). *Let Q be a query to an abductive logic program $\langle T, IC, Ab \rangle$. If (D', σ) is an answer to Q, then there exists a derivation starting from $Q \wedge IC$ of a formula containing a nonfailure leaf node from which an answer (D, σ) to Q can be extracted such that D is a subset of D'.*

Theorems 1 and 2 hold when $\models$ is logical consequence in three-valued logic [12]. Theorem 1 continues to hold when $\models$ is logical consequence in two-valued logic, but Theorem 2 needs to be weakened, in which case a form of refutation completeness can also be shown:

Theorem 3 (Completeness for two-valued logic). *Let Q be a query to an abductive logic program $\langle T, IC, Ab \rangle$, where T is call-consistent (see [12]). Suppose there is a derivation starting from $Q \wedge IC$ of a formula F_n, all of whose nodes are leaf nodes.*

1. *If (D', σ) is an answer to Q (in two-valued logic), then there exists a node in F_n from which an answer (D, σ) to Q can be extracted such that D is a subset of D'.*
2. *If $T \cup CET \cup IC \models \neg Q$ (in two-valued logic), then all nodes in F are failure nodes.*

The proof of Theorem 2 in [6] is based upon a theorem of Kunen [12] which shows that, for every three-valued logical consequence C of a logic program T in iff form, there exists a bottom-up derivation of C by means of the three-valued immediate consequence operator associated with T. Our proof constructs a top-down derivation by means of the iff proof procedure, using the bottom-up derivation of Kunen's theorem as a guide.

The top-down derivation is constructed in two phases. In the first phase, defined atoms occurring directly as conjuncts or occurring in the conditions of implications in a node are reduced to abducible atoms, using unfolding and splitting. Equalities are simplified by using rewrite rules and case analysis wherever possible.

In the second phase, propagation and factoring are applied, again using equality rewrite rules and case analysis wherever possible. The two phases are repeated until no further inference rules are applicable.

The proof of Theorem 2 shows only that there exists a derivation constructed by means of the two-phase process described above. It does not show that there exists a derivation for any inference rule application strategy. However, based upon our experience with applying the proof procedure and based upon the analogy with SLDNF, we believe that the proof procedure is complete for any "well-behaved" (e.g., fair) strategy. Moreover, it seems likely that the existing completeness proof can be strengthened for this purpose.

6. CONCLUSIONS

6.1. Related Work

Except for [1], all other proof procedures for abductive logic programming (e.g., [17, 2, 3, 7, 8]) use the if form for definitions and skolemization instead of implicit or explicit existential quantification. The proof procedure in [1], however, is not fully defined for nonpropositional programs. Thus, the iff proof procedure can be regarded as an extension of the proof procedure of [1] to the nonpropositional case.

In many respects, the iff proof procedure is closest to SLDNFA [2,3]. Recent versions of SLDNFA, moreover, also use variables instead of skolemization. However, SLDNFA uses the if form for definitions where we use the iff form. Other differences are that: 1) SLDNFA uses a weak safety rule where we use admissibility, 2) SLDNFA uses a special form of "negative resolution" where we use propagation, 3) SLDNFA does not have an explicit case analysis rule, but obtains similar results by having several ways in which answers can be extracted from derivations, and 4) SLDNFA modifies the unification algorithm where we employ rewrite rules for equality. We believe that the choices we have made result in a simpler proof procedure. Moreover, we can prove a stronger completeness result (Theorem 2). On the other hand, SLDNFA constructs more general answers containing variables.

6.2. *Future Work*

As indicated in the discussion of the proof of Theorem 2 at the end of Section 5, the theorem needs to be strengthened to show that any appropriately "well-behaved" strategy for applying the inference rules will generate all minimal answers to a query. It is also important to investigate which strategies are more efficient than others. For example, it seems that in most cases, propagation should be applied with higher priority than unfolding, and that splitting should be delayed as long as possible.

The proof procedure, with little alteration, can be applied to more general forms of definitions and integrity constraints. In particular, definitions

$$P(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n$$

can be allowed to contain implications as conjuncts of the disjuncts D_i. Integrity constraints can be allowed to contain existentially quantified variables in the conclusion. Moreover, it also seems possible to remove or greatly relax the allowedness restriction.

It may be that the proof procedure can be further simplified and more informative answers can be obtained by changing the semantics. This possibility is investigated in [10, 11, 14], where the abducible predicates in a nonfailure leaf node are not completed, and the integrity constraints are required to be consistent with the answer rather than to be theorems of the completed definitions. Like SLDNFA, the proof procedure of [10, 11, 14] constructs more general answers than the iff proof procedure.

We are grateful to Phan Minh Dung, Fariba Sadri, Francesca Toni, and Gerhard Wetzel for comments on earlier drafts of this paper.

REFERENCES

1. Console, L., Theseider Dupre, D., and Torasso, P., On the relationship between abduction and deduction, *J. Logic and Computation* 2(5):661–690 (1991).
2. Denecker, M. and De Schreye, D., SLDNFA: An abductive procedure for normal abductive programs, in: *Proc. ICSLP*, MIT Press, 1992, pp. 868–700.

3. Denecker, M., Knowledge representation and reasoning in incomplete logic programming, Ph.D. thesis, Department of Computer Science, K.U. Leuven, Belgium, 1993.
4. Fung, T. H., A modified abductive framework, in: *Proc. Logic Programming Workshop, WLP'94*, N. Fuchs and G. Gottlob (eds.), 1994.
5. Fung, T. H., Abduction with Clark completion, in: *Proc. ICLP*, MIT Press, 1995.
6. Fung, T. H., Abduction by deduction, Ph.D. thesis, Imperial College, University of London, England, 1996.
7. Kakas, A. C. and Mancarella, P., *Abductive Logic Programming*, 1990.
8. Kakas, A. C. and Mancarella, P., Database updates through abduction, in: *Proc. 16th VLDB*, Morgan Kaufmann, Los Altos, CA, 1990, pp. 650–661.
9. Kowalski, R. A., Logic programming in artificial intelligence, in: *Proc. IJCAI*, 1991.
10. Kowalski, R. A., Toni, F., and Wetzel, G., Towards a declarative and efficient glass-box CLP language, in: *Proc. Logic Programming Workshop, WLP'94*, N. Fuchs and G. Gottlob (eds.), 1994.
11. Kowalski, R. A., Wetzel, G., and Toni, F., A unifying framework for ALP, CLP and SQO, Department of Computing, Imperial College, London, England, Apr. 1996.
12. Kunen, K., Negation in logic programming, *J. Logic Programming* 4:231–245 (1987).
13. Martelli, A. and Montanari, U., An efficient unification algorithm, *Trans. Programming Languages and Syst.* 4(2):258–282 (1982).
14. Wetzel, G., Kowalski, R. A., and Toni, F., A theorem-proving approach to CLP, in: *Workshop Logische Programmierung*, A. Krall and U. Geske (eds.), GMD-Studien Nr. 270, Sept. 1995, pp. 63–72.
15. Kowalski, R., Using meta-logic to reconcile reactive with rational agents, in: *Meta-Logics and Logic Programming*, K. Apt and F. Turini (eds.), MIT Press, 1995.
16. Kowalski, R. and Sadri, F., Towards a unified agent architecture that combines rationality with reactivity, in: *Proc. Workshop on Logic in Databases (LIDS'96)*, San Miniato, Italy, Springer-Verlag, 1996.
17. Eshghi, K. and Kowalski, R., Abduction through deduction, Department of Computing, Imperial College, London, England, 1988.
18. Kakas A., Kowalski, R., and Toni, F., Abductive logic programming, *J. Logic and Computation* 2(6):719–770 (1993).