Theoretical Computer Science 54 (1987) 3–28
North-Holland

AUTOMATIC INDUCTIVE THEOREM PROVING USING PROLOG

Jieh HSIANG and Mandayam SRIVAS

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794, U.S.A.

Abstract. Although PROLOG is a programming language based on techniques from theorem proving, its use as a base for a theorem prover has not been explored until recently (Stickel, 1984). In this paper, we introduce a PROLOG-based deductive theorem proving method for proving theorems in a first-order inductive theory representable in Horn clauses. The method has the following characteristics:

(1) It automatically partitions the domains over which the variables range into subdomains according to the manner in which the predicate symbols in the theorem are defined.

(2) For each of the subdomains the prover returns a lemma. If the lemma is **true**, then the target theorem is true for this subdomain. The lemma could also be an induction hypothesis for the theorem.

(3) The method does not explicitly use any inductive inference rule. The induction hypothesis, if needed for a certain subdomain, will sometimes be generated from a (limited) forward chaining mechanism in the prover and not from employing any particular inference rule.

In addition to the backward chaining and backtracking facilities of PROLOG, our method introduces three new mechanisms—*skolemization by need*, *suspended evaluation*, and *limited forward chaining*. These new mechanisms are simple enough to be easily implemented or even incorporated into PROLOG. We describe how the theorem prover can be used to prove properties of PROLOG programs by showing two simple examples.

1. Introduction

PROLOG is a powerful and versatile programming language based on theorem proving techniques such as unification and resolution. Many of its implementations perform inferences at a much higher speed than general purpose theorem provers. Despite this fact PROLOG has not been successfully used as a theorem prover. Some of the reasons for this are that PROLOG is restricted in expressive power (Horn clause based), and has other obstacles such as the lack of occur checks and the inability to prove properties in which variables are universally quantified to range over recursively constructed domains.

Stickel (in [26]) gave an approach for using PROLOG as a general theorem prover. It included a general inference rule which provides PROLOG with the ability of dealing with non-Horn clauses (and, consequently, does not use the closed world assumption [23]), as well as mechanisms for occur checks.[1] The universal quantification problem was handled in [26] in a standard way—by skolemizing all of

[1] The occur check problem was discussed in detail in [22].

the universally quantified variables right at the beginning of the proof process. In [20] Lloyd and Topor show the soundness of the negation as failure rule and SLDNF-resolution for *extended* logic programs in which the body of a clause (and a query) can be an arbitrary first-order formula. They show how the increased expressibility of extended programs and goals can be implemented in a PROLOG system with a sound implementation of negation as failure rule. In [20] universal quantification is handled by *completing* the extended programs by adding a *closure* rule which lists the constants belonging to the domain of the universally quantified variables. Neither of the above approaches, however, works for proving inductive theorems, i.e., theorems which are only true in the initial model defined by the Horn clauses. Such properties often arise when proving properties of programs defined on recursive data structures such as *Lists*.

In this paper we introduce an inductive theorem proving method to provide a more satisfactory answer to the above problem. This is done by supplementing the backward chaining mechanism of PROLOG with three new mechanisms—*skolemization by need, suspended evaluation*, and *limited forward chaining*. The new mechanisms introduced are simple enough to be easily implemented or even incorporated into PROLOG. We demonstrate the use of the theorem prover for verifying PROLOG programs and proving properties of data types.

Our theorem proving method has the following characteristics.

(1) It can be used for proving inductive properties in which the variables range over recursively constructed domains.

(2) It automatically divides the domain(s) of the universally quantified variable(s) into a finite number of subdomains. Each subdomain is characterized by the instantiations (for the quantified variables) that the theorem prover returns.

(3) It proves the validity of the proposition for each subdomain separately. For each subdomain, it returns a *Lemma* the validity of which guarantees the validity of the proposition for that subdomain. The *Lemma* could be the literal **true**, or the induction hypothesis for the subdomain, or an arbitrary (Horn clause) formula. In the first two cases the *Lemma* is already proved, in the third case the *Lemma* can be fed back (by the user) to the prover again. Thus, the method does not use any explicit inductive inference rule.

(4) Also, since the method is based on subgoal reduction, the prover *always terminates*.

The method has the following restrictions. The proposition to be proved must be in the form of a Horn clause in which the antecedents are arbitrary literals, but the consequent is a PROLOG equality predicate. The second restriction is that every $(n+1)$ary predicate $P(x_1, \ldots, x_n)$ that appears in the proposition is defined as a total n-ary function with respect to its last argument. There are methods [4] to check the second condition.

One way of building the theorem prover is to integrate it into the environment provided by a PROLOG system. The extended environment, besides inheriting the normal features of PROLOG, will incorporate all the new mechanisms proposed in

the paper. In such an environment a universally quantified proposition prop($\bar{X}$) is proved by typing it in as a query. A simpler way of building the theorem prover is to implement it as a predicate on top of a PROLOG interpreter. In such a case a property is proved by defining it as a set of PROLOG clauses, and then invoking the theorem proving predicate. A preliminary implementation of our method has been completed using the second approach. We maintain the perspective implied by the second approach in describing the method in this paper, as well.

1.1. Organization of the paper

The rest of the introduction provides an overview of the theorem proving method. Section 2 gives a formal description of the theorem prover. Section 3 describes in detail all the new mechanisms used by the theorem proving method. Section 4 gives an algorithm for implementing the method. (This section may be skipped at first reading especially if the reader is not interested in the details of the algorithm). Section 5 provides a discussion of how the output produced by the prover is to be interpreted, and a comparison of our method with other inductive theorem proving methods. Appendix A gives an illustration of the method on a couple of examples.

1.2. An overview

We begin by describing the problems encountered in using PROLOG for proving an inductive property by simply typing the property as a query into a PROLOG interpreter. We introduce the new mechanisms by illustrating how to overcome these problems, and then present the method informally.

1.2.1. The problem of unbounded depth-first search strategy

PROLOG, which uses a query-respond paradigm to communicate with its users, is capable of proving only existentially quantified properties. For example, consider proving the associativity property of the 'append' of two lists.

$$\forall X, Y, Z, L1, \ldots, L4[\text{append}(X, Y, L1), \text{append}(L1, Z, L2), \\ \text{append}(Y, Z, L3), \text{append}(X, L3, L4) \supset L2 = L4].$$

'Append' is defined as a predicate which checks whether its third argument is the concatenation of its first two arguments. The property is expressed as a PROLOG clause defining a predicate prop(X, Y, Z) as follows:[2]

```
append([ ], L, L)
append([A|L1], L2, [A|L3]):- append(L1, L2, L3),

prop(X, Y, Z):- append(X, Y, L1), append(L1, Z, L2)
                append(Y, Z, L3), append(X, L3, L4), L2 = L4.
```

[2] The clause defining 'prop' and the property to be proved are not logically equivalent. However, the manner in which the goals are processed by our method makes this discrepancy inconsequential. More about this is discussed in Section 2.1.

The property cannot be proved by executing the query:

$$:-\text{prop}(X, Y, Z)$$

since PROLOG will only provide an instance (not necessarily the most general one) of the *input* variables X, Y, and Z which satisfies the property. In some interpreters of PROLOG (such as the C-PROLOG [3]), it is possible to request another instantiation (if any) which also satisfies prop(X, Y, Z). Such a feature will not help in general when the domains of the variables are infinite.

In order to deal with the problem of infinite domains, we use the notion of skolemization employed in refutational theorem proving. Henceforth a *hatted variable*, such as $\hat{X}$, will be used to denote a skolem constant. A skolem constant is like a metavariable since it can denote an arbitrary value belonging to the domain under consideration. However, directly executing a (skolemized) goal such as

$$:-\text{prop}(\hat{X}, \hat{Y}, \hat{Z})$$

will not lead to a proof since the first subgoal, append($\hat{X}$, $\hat{Y}$, $L1$) does not match with any clause head. To solve the above problem we introduce a new concept called *Ω-satisfiability*, which is weaker than the standard notion of satisfiability of a goal in PROLOG. Ω-satisfiability gives us a way of handling unsatisfiable goals whose unsatisfiability is due to the appearance of skolem constants. For instance, consider the goal append($\hat{Y}$, $\hat{Z}$, $L3$). The goal append($\hat{Y}$, $\hat{Z}$, $L3$) is not satisfiable since $\hat{Y}$ unifies with neither [ ] nor $[A|L]$. However, we know that this goal *should* be satisfiable since $\hat{Y}$, being a list, has to be either [ ] or $[A|L]$ for some A and L. Therefore, we *suspend the evaluation* of this goal by treating it as being satisfied. The suspended state of the goal is recorded by binding $L3$ to a closure, called *Ω-binding*.[3] The Ω-binding contains the constraint that $L3$ has to meet for the goal to succeed. In our notation, $L3$ is bound to a term of the form $\Omega(l3{:}\text{append}(\hat{Y}, \hat{Z}, l3))$, where append($\hat{Y}$, $\hat{Z}$, $l3$) is called the *Ω-constraint* of $L3$, and $l3$ is called the (bound) Ω-variable. A goal which can be satisfied in this way is called *Ω-satisfiable*.

To fully characterize the notion of Ω-satisfiability, we need to take into consideration two other situations in which a goal that would normally fail (under PROLOG satisfiability) would have to succeed for our purpose. Both these situations occur when the failure of a goal is because some of its arguments have Ω-bindings. To see the first situation, consider the goal append($L1$, $\hat{Z}$, $L2$), where $L1$ is bound to $\Omega(l1{:}\text{append}(\hat{X}, \hat{Y}, l1))$. We make this goal succeed by once again suspending the evaluation of this goal, and generating an Ω-binding for $L2$. Note that this goal has to be added as a new constraint to the currently existing Ω-bindings in the goal, and all the constraints in the currently existing Ω-bindings should be *propagated* to the new bindings generated. Thus, $L1$ is bound to $\Omega(l1{:}\text{append}(\hat{X}, \hat{Y}, l1)$,

[3] A similar notion has also been used by Kornfeld [16] for enriching the unification to include equational axioms.

append($l1$, $\hat{Z}$, $l2$)) and $L2$ is bound to $\Omega(l2$:append($\hat{X}$, $\hat{Y}$, $l1$), append($l1$, $\hat{Z}$, $l2$)). The second situation is illustrated by the following example. Consider the goal append([ ], $L3$, $L2$), where $L3$ is bound to $\Omega(l3$:append($\hat{Y}$, $\hat{Z}$, $l3$)), and $L2$ is bound to $\Omega(l2$:append($\hat{Y}$, $\hat{Z}$, $l2$)). We would want this goal to succeed (although it would normally fail) because the bindings of $L2$ and $L3$ although structurally distinct impose the same constraint on the variables. We fix this problem by using *Ω-equivalence* instead of PROLOG equivalence, which is structural identity, while comparing terms. Two Ω-terms are *Ω-equivalent* if they can be made identical upto renaming of the Ω-variables. On all other terms Ω-equivalence behaves just as PROLOG equivalence.

By using Ω-satisfiability instead of ordinary PROLOG satisfiability the query:

$$:-\text{prop}([\ ],\ \hat{Y},\ \hat{Z}) \tag{1}$$

can be executed successfully. The constraint (in the Ω-bindings generated during the execution of 'prop') on which the success of the query depends is append($\hat{Y}$, $\hat{Z}$, l). This constraint is guaranteed to be satisfied because (by our totality assumption about the predicates) there always exists such an l for arbitrary lists that $\hat{Y}$ and $\hat{Z}$ denote. Thus, the successful execution of the above query proves the associativity property of 'append' for the case where X is [ ].

1.2.2. The problem of induction

As illustrated above, if we use the notion of suspended evaluation in executing an appropriately skolemized goal, we would, in general, be left with an Ω-constraint at the end of a successful execution of the goal. The Ω-constraint is sometimes obviously true, as was the case in the situation shown above. But, sometimes it might denote a conjunction of goals that is implied by a smaller instance of the original goal that we were trying to satisfy. It is extremely useful to detect this situation since it could give us the induction hypothesis needed to complete the inductive step in the proof of the original goal. PROLOG cannot detect such an implication since it uses only *backward chaining* (deducing subgoals from a goal), but not *forward chaining* (deducing a goal from a set of subgoals). It is not hard, in principle, to incorporate general forward chaining into PROLOG by constantly checking the remaining of the subgoals to see if some of them satisfy a clause. But this is undesirable since it would be extremely inefficient. We deal with this problem by introducing a *limited* forward chaining mechanism. This consists of

(1) using only the clause that describes the proposition to be proved for forward chaining, and

(2) only attempting to perform forward chaining on the Ω-constraint obtained at the end of Ω-satisfying all the goals in the body of the proposition clause.

For instance, consider the execution of the query:

$$:-\text{prop}([\hat{A}\,|\,\hat{L}],\ \hat{Y},\ \hat{Z}). \tag{2}$$

Assuming we are using Ω-satisfiability, the above query can be successfuly executed.

At the end of the execution the Ω-bindings generated will have the following constraint:

$$\text{append}(\hat{L}, \hat{Y}, l1), \text{append}(l1, \hat{Z}, l2), \text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{L}, l3, l4), l2 = l4.$$

By performing forward chaining on the above set of goals using the clause defining 'prop', we have $\text{prop}(\hat{L}, \hat{Y}, \hat{Z})$ as the constraint on which the validity of $\text{prop}([\hat{A} \mid \hat{L}], \hat{Y}, \hat{Z})$ is dependent. Combining the execution of the queries (1) and (2), we have established the validity of the following formulas which completes the proof of $\forall X, Y, Z\ \text{prop}(X, Y, Z)$:

(1) $\forall Y, Z\ \text{prop}([\ ], Y, Z]$,

(2) $\forall A, L, Y, Z\ (\text{prop}(L, Y, Z) \supset \text{prop}([A \mid L], Y, Z))$.

Note that it might not always be possible to perform forward chaining on the Ω-constraint generated. In such a case the constraint is merely returned as a *Lemma* that has to be proved for establishing the validity of the proposition.

1.2.3. The problem of skolemization

In completing the proof (shown above) of 'prop', the skolem constants with which the variables X, Y, and Z were instantiated were chosen a priori. In the first case it was chosen to be $([\ ], \hat{Y}, \hat{Z})$, and in the second case it was $([\hat{A} \mid \hat{L}], \hat{Y}, \hat{Z})$. Together they form a *complete* set of skolemizations because they completely span the domain under consideration, namely the triple product of *List*. While completeness of the set of skolemization is certainly necessary, it is also equally important to have the right kind of partitioning of the domain. For instance, the naive skolemization, such as $(\hat{X}, \hat{Y}, \hat{Z})$ in the above example, in which every universally quantified variable is skolemized to a distinct unstructured constant is obviously complete but rarely leads to an inductive proof. For instance, execution of the query:

$$:- \text{prop}(\hat{X}, \hat{Y}, \hat{Z}) \tag{3}$$

will give $\text{prop}(\hat{X}, \hat{Y}, \hat{Z})$ as the *Lemma* leading us back to where we began.

The skolemization that is likely to lead to a proof is dependent on the inductive structure of the definition of the predicates in the proposition, and on the structure of the terms constructing the domain. To automate the generation of skolem constants in a way that takes into account the inductive structure, we introduce a mechanism called *skolemization by need.* In our theorem proving method bindings for the universally quantified variables are generated using the skolemization by need mechanism.

Under this method, the universally quantified variables in a query start out as free variables (i.e., like any other variables in PROLOG) instead of being replaced by skolem constants (as was done in the proof of 'prop' shown above). This allows PROLOG unification to keep instantiating them until they are skolemized. A variable gets *skolemized* only (and immediately) after a *decision* about the value to be bound for that variable is made. We consider a decision to have been made when

(1) the variable is unified with a nonvariable term, or

(2) the variable appears in a goal whose execution does not lead to any new subgoals.

Note that the second situation may happen either because the goal was successfully matched with a fact, or because its evaluation was suspended by generating Ω-bindings. The fact that a variable X is skolemized is indicated by replacing every free variable in the term currently bound to X by its corresponding hatted skolem constant.

For instance, the goal append(X, Y, $L1$) where X and Y are universally quantified, would succeed when unified with the fact in the 'append' program; X would be bound to [ ] and Y (and hence also $L1$) would be bound to a skolem constant $\hat{Y}$ (rather than a variable) since the goal does not generate any new subgoals. On the other hand if append(X, Y, $L1$) were unified with the head of the second clause in the program for 'append', then X would get instantiated to $[A|L]$. X would then immediately get skolemized to $[\hat{A}|\hat{L}]$ since $[A|L]$ is a nonvariable term. The variable Y is not skolemized at this point since no decision needs to be made about its value; neither is $L1$ since it is not an input variable.

When skolemization by need is used in conjunction with Ω-satisfiability in executing a query, the response will not only be a *Lemma* (constructed from the Ω-constraint), but also the skolemization which was responsible for the *Lemma*. To obtain a complete set of skolemizations and the corresponding *Lemmas*, we use the backtracking mechanism of PROLOG by forcing a failure after an execution of the query. The totality restriction we impose on the definition of predicates guarantees that a complete set of skolemizations will be produced after a finite number of forced failure attempts of the query. (A justification of this is provided in Section 3.5). For instance, the first execution of the query :– prop(X, Y, Z) would yield the skolemization ([ ], $\hat{Y}$, $\hat{Z}$) and the lemma **true**. A forced failure would yield, after backtracking, the skolemization ($[\hat{A}|\hat{L}]$, $\hat{Y}$, $\hat{Z}$) and the *Lemma* prop($\hat{L}$, $\hat{Y}$, $\hat{Z}$).

1.2.4. An outline of the method

The properties $(\forall \bar{X}\phi(\bar{X}))$ that our method is capable of proving have the general form of Horn clauses:

$$\phi(\bar{X}): \quad \forall \bar{Z}(P_1(\bar{X}, \bar{Z}) \wedge \cdots \wedge P_n(\bar{X}, \bar{Z}) \supset Q(\bar{X}, \bar{Z}))$$

where the P_i's (the *antecedents*) are arbitrary predicates and Q (the *consequent*) is a literal representing some equality $s = t$. For convenience we shall use $Q(\bar{X}, \bar{Z})$ for the consequent throughout the paper instead of the literal $s = t$, even though Q is not a predicate symbol. We assume that $\bar{X}$ and $\bar{Z}$ (lists of variables) are the only variables in the predicates, and that every variable in $\bar{Z}$ appears in at least one of the antecedents.

Our method consists of Ω-satisfying every P_i subjecting every variable in $\bar{X}$ to skolemization by need in the process. If any of the P_i's cannot be Ω-satisfied, then the proposition is vacuously true because one of the antecedents is false. The mechanisms of Ω-satisfaction and skolemization by need (and the fact that every

variable in $\bar{Z}$ appears in at least one of the antecedents) guarantee that no variable in $\bar{X}$ or $\bar{Z}$ remains free at the end of Ω-satisfying all the antecedents. More specifically, the following conditions are guaranteed:

(i) Every variable in $\bar{X}$ is skolemized.

(ii) No variable in $\bar{Z}$ is free; it is either bound to an Ω-term, or to a term that contains skolemized variables.

After processing the antecedents, an attempt is made to Ω-satisfy $Q(\bar{X}, \bar{Z})$. Since none of the variables in Q is free, the outcome of such an attempt can be one of the following:

(1) Q is satisfied under PROLOG satisfiability (with PROLOG equality extended to Ω-equivalence). In this case no new Ω-bindings are generated, nor any of the existing Ω-bindings are altered.

(2) Q is Ω-satisfied by adding new constraints to the existing Ω-bindings. This happens when Q cannot be satisfied as in (1). In this case, according to the Ω-satisfiability mechanism, all the constraints in the current bindings will be merged, and Q will be included into the Ω-bindings as a new constraint. Thus, when Q cannot be satisfied as in (1) it will always become a part of the Ω-constraint generated. In the former case the proposition is proved for the present skolemization because the validity of the consequent was shown despite the constraints on which the validity of the antecedents is based. Hence, the literal **true** is returned as the *Lemma*. In the latter case, the validity of the proposition is dependent on the Ω-constraint. Note that the eventual Ω-constraint, although represented as a list of goals, itself denotes a Horn clause formula with the constraints generated from the P_i's forming the antecedents and the constraint generated from Q forming the consequent. This formula is returned as the *Lemma* after checking whether it is an instance of ϕ.

In both the cases, the skolemization generated for the variables in $\bar{X}$ is returned along with the *Lemma*. This takes care of the proof for the partition of the domain (of $\bar{X}$) that is characterized by the skolemization. To complete the proof for the remaining parts of the domain, it is necessary to backtrack (*undoing* the skolemizations in the process), and re-Ω-satisfy the goals in the proposition. The backtracking needed here is much like the one used by PROLOG, and will be explained later in the paper.

2. Functional description of the prover

2.1. Representation of the proposition

The PROLOG data base that the theorem prover will operate on should include a description of the proposition to be proved, and a complete definition of all the predicates used in the proposition. In our formalism the proposition $\phi(\bar{X})$

$$\phi(\bar{X}): \quad \forall \bar{Z}(P_1(\bar{X}, \bar{Z}) \wedge \cdots \wedge P_n(\bar{X}, \bar{Z}) \supset Q(\bar{X}, \bar{Z}))$$

to be proved is represented as a predicate prop($\bar{X}$) defined by the following PROLOG

clause:

$$\text{prop}(\bar{X}) :- P_1(\bar{X}, \bar{Z}), \ldots, P_n(\bar{X}, \bar{Z}), Q(\bar{X}, \bar{Z}).$$

As a convention we refer to the variables in $\bar{X}$ as *input variables*. Note that ϕ is not logically equivalent to the PROLOG clause. However, this discrepancy does not have any ill effect because 'prop' is only used as a means of representing the property to be proved, not as a predicate in any other clauses. Also, our method requires the antecedents be processed before the consequent. The left-to-right strategy used by PROLOG for processing *and* goals accomplishes this automatically when ϕ is represented as 'prop'. 'prop' thus defined also makes it convenient to check if the Ω-constraint generated is the inductive hypothesis by using this clause to perform forward chaining.

2.2. Description of the arguments

For ease of presentation, we introduce a new predicate ind_prove(*Theorem*, *Premise*, *Lemma*) to serve as the prover. To prove a proposition prop($\bar{X}$), the prover is invoked by ind_prove(prop($\bar{X}$), *P*, *L*). This would result in a single skolemization (in the form of instantiations for the variables in $\bar{X}$), and a value for *P* and *L*. The rest of the skolemizations, and their correspondng premises and lemmas may be obtained through PROLOG backtracking.

The arguments of ind_prove are described below:

Theorem is the proposition to be proved.

Premise is a list of conditions (predicates) on the skolem constants appearing in the skolemization generated for $\bar{X}$. When the list is empty, *Premise* is considered to be **true**; otherwise, it is considered as the conjunction of all the predicates in the list. A nontrivial *Premise* appears mostly when some predicate is defined conditionally.

Lemma is also a list of conditions (or **true**) like *Premise*. These conditions determine the validity of the proposition being proved for the corresponding skolemization. The conditions in *Lemma* are interpreted in a way different from the *Premise*. If the list of conditions in *Lemma* is $(A_1, \ldots, A_m, Q')$, where Q' is an instance of the consequent Q, then its logical meaning is

$$A_1 \wedge \cdots \wedge A_m \supset Q'.$$

Suppose $\{\bar{X}_1, \ldots, \bar{X}_k\}$ is the set of all instantiations generated for $\bar{X}$ by repeated invocations of ind_prove(prop($\bar{X}$), *Premise*, *Lemma*). For each $\bar{X}_j$, let $\text{Pr}_j(\bar{X}_j)$ be the premise and $\text{Lem}_j(\bar{X}_j)$ be the lemma produced by ind_prove. Let $D_{\bar{X}}$ denote the domain of values, i.e., ground terms, over which $\bar{X}$ ranges. Then the outputs produced by ind_prove automatically satisfy the following conditions:

(1) *Well-spannedness*: For every ground $\bar{d} \in D_{\bar{X}}$, there is some j such that $\bar{d}$ is an instance of $\bar{X}_j$ and $\text{Pr}_j(\bar{d})$ is true. In other words, the set of instantiations *well-span* (cf. [10, 25]) domain $D_{\bar{X}}$. The use of *Premise* is for the possible splitting of cases in the domain. For example, given a proposition with two inputs *A* and *L* where

A is an atom and L is a list, the set of instantiations may be $(A, [\ ])$, $(A, [B|L])$, and $(A, [B|L])$, with premises respectively, **true**, $A = B$, and $A \neq B$.

(2) *Problem reduction*: For each $\bar{d} \in D_{\bar{X}}$ and for the proper instantiation and premise index j for which $\bar{d}$ satisfies (1), we have the property:

$$\text{if } \mathrm{Lem}_j(\bar{d}), \quad \text{then } \mathrm{prop}(\bar{d}).$$

Property (1) above describes a proper set of instantiations and property (2) indicates that, for a particular instance $\bar{d}$, $\mathrm{Lem}_j(\bar{d})$ has to be proved for $\mathrm{prop}(\bar{d})$ to be true.

For the 'append' example, ind_prove would generate the following set of instantiations with the corresponding lemmas:

Instantiation	*Premise*	*Lemma*
$([\ ], Y, Z)$	**true**	**true**
$([A|L], Y, Z)$	**true**	$\mathrm{prop}(L, Y, Z)$

This means that

(1) $\forall Y, Z\ \mathrm{prop}([\ ], Y, Z)$,

(2) $\forall A, L, Y, Z(\mathrm{prop}(L, Y, Z) \supset \mathrm{prop}([A|L], Y, Z))$.

Since $\mathrm{prop}(L, Y, Z)$ is the induction hypothesis of $\mathrm{prop}([A|L], Y, Z)$, we have proved $\forall X, Y, Z\ \mathrm{prop}(X, Y, Z)$, and therefore 'append' is associative.

3. The new mechanisms

This section gives a detailed description of all the newly introduced mechanisms which form the building blocks for our theorem proving method.

3.1. Skolemization by need

This mechanism provides a systematic way of skolemizing the input variables in the proposition.This method of skolemization is distinguished from the conventional methods in that the variables are not skolemized at the start of the resolution process. Instead, every input variable is left unskolemized (so that it can get instantiated by the unification of PROLOG) until a *decision* about its value has to be made. At this point the input variable is skolemized by simply *hatting* every variable occurring in the term currently bound to the input variable. We consider such a *decision* about an input variable to have been made when

(1) the variable is unified with a nonvariable term, or

(2) the variable appears in a goal whose execution does not lead to any new subgoal. Note that this situation may occur either because the goal was successfully matched with a fact, or because its evaluation was suspended for generating Ω-bindings.

As an example, if the first subgoal append(X, Y, $L1$), where X and Y are input variables, is unified with the head of the second clause in the definition of 'append' (Section 1.2.1), append($[A|L1]$, $L2$, $[A|L3]$), then the variable X will be instantiated to $[A|L]$. Then, X would be immediately skolemized to $[\hat{A}|\hat{L}]$ by hatting the variables A and L. The variable Y is not skolemized yet because no decision about its value is made. On the other hand, let us suppose the input variable Y appears in a goal append($L2$, Y, $L3$), in which $L2$ has an Ω-binding (see next section). Then this goal would be Ω-satisfied without creating any subgoals, and hence Y would be skolemized to $\hat{Y}$.

3.2. Ω-Satisfiability and Ω-binding

Ω-satisfiability is a notion of satisfiability weaker than the one used in PROLOG. It is used primarily to handle the failure of a goal that arises because some of the variables in the goal are skolemized. We define Ω-satisfiability so that a goal would succeed in such a situation by generating a special kind of binding, called an *Ω-binding*, for the free variables in the goal. Before describing how Ω-bindings are generated we need to introduce some definitions.

An *Ω-term* is a term of the form $\Omega(x{:}P)$ where x is a variable, called the *Ω-variable*, and P is a (conjunction of) predicate(s), called the *Ω-constraint*. A variable X is *Ω-bound* to t if t is an Ω-term or it is a term containing an Ω-term as a subterm. If a variable X is bound to $[a|\Omega(l{:}P(\hat{Y}, z), Q(z, l))]$, it means that X is bound to the list $[a|L]$ where L is a list satisfying $P(\hat{Y}, Z) \wedge Q(Z, L)$ for some Z. (Note that in our method an input variable will never be Ω-bound.)

The purpose of Ω-binding is to treat certain unsatisfiable goals as constraints, which may eventually become part of the *Lemma* or *Premise*. From now on we use the word Ω-term loosely to mean either an Ω-term as defined above or any term that contains an Ω-term.

Definition. Two Ω-terms t_1 and t_2 are *Ω-equivalent* (denoted as $t_1 =_\Omega t_2$) if they are identical upon renaming of the Ω-variables.

For example, $1+\Omega(n{:}P(\hat{Y}, n))$ and $1+\Omega(m{:}P(\hat{Y}, m))$ are Ω-equivalent, while $1+\Omega(n{:}P(\hat{Y}, n))$ and $1+\Omega(m{:}Q(\hat{Y}, m))$ are not (even if P and Q can be proven to be equivalent by other means), nor are $1+\Omega(n{:}P(\hat{X}, n))$ and $1+\Omega(m{:}P(\hat{Y}, m))$.

Definition. Let $P(t_1, \ldots, t_n)$ be an arbitrary goal, where the t_i's are either Ω-terms, constants, skolem constants, or free variables. Then, P is *Ω-satisfiable* if one of the following holds:

(1) P unifies with a clause head provided a skolemized variable in P is allowed to be instantiated.

(2) Condition (1) does not hold, but P can be satisfied in PROLOG with the PROLOG equality extended to include Ω-equivalence.

(3) The goal cannot be satisfied as in (2), and at least one of the variables in P is skolemized or Ω-bound.

For example, if "$G(X,[a|X])$." is a PROLOG fact, then the goal $G(\Omega(m{:}P(\hat{Y},m)),[a|\Omega(n{:}P(\hat{Y},n))])$ is Ω-satisfiable since the goal and the fact match and $\Omega(m{:}P(\hat{Y},m)) =_\Omega \Omega(n{:}P(\hat{Y},n))$. Note that conditions (1) and (3) above represent situations in which a potential failure of a goal is caused because of the existence of a skolemized variable in the goal.

3.2.1. Ω-binding generation

Ω-bindings will occur as a 'side-effect' when a goal $P(t_1,\ldots,t_n)$ gets Ω-satisfied as per the conditions (1) and (3) in the definition above. In such a case, the binding of every non-input variable in $P(t_1,\ldots,t_n)$ will be changed according to the following rules. (Note that skolemization by need requires that all unskolemized input variables be skolemized at such a juncture.) The Ω-binding generated for a variable indicates the constraint that the variable ought to satisfy.

Case 1: At least one argument of $P(t_1,\ldots,t_n)$ is a free non-input variable. There are two subcases. Without any loss of generality, let us assume that t_n is the only free non-input variable X. If there are more than one free non-input variables in P, then a similar action will take place on each of them.

Case 1(a): None of the arguments of $P(t_1,\ldots,t_{n-1},X)$ is Ω-bound. In this case we consider the goal $P(t_1,\ldots,t_{n-1},X)$ satisfied by simply binding X to the Ω-term $\Omega(x{:}P(t_1,\ldots,t_{n-1},x))$. Every non-input free variable in P is bound to an Ω-term in a similar way. It is not hard to see that each Ω-term generated here will have the same Ω-constraint.

Case 1(b): Some arguments of $P(t_1,\ldots,t_{n-1},X)$ are already Ω-bound. In this case, X is bound to an Ω-term which is obtained by *merging* the Ω-terms of all the Ω-bound variables of P. Without any loss of generality, let us assume $t_1,\ldots,t_k$ are free input variables and each of $t_{k+1},\ldots,t_n$ is Ω-bound to a term of the form $\Omega(y_j{:}C_j)$. Then, X is bound to the term $\Omega(x{:}C)$, where the constraint C is

$$C_{k+1}\cup\cdots\cup C_n\cup\{P(t_1,\ldots,t_k,y_{k+1},\ldots,y_n,x)\}.$$

For example, consider $P(\Omega(u{:}R(u,\hat{Y})),[\hat{A}|\Omega(l{:}Q(\hat{Z},l))],X)$ (that is, a predicate $P(U,[\hat{A}|L],X)$ with U and L bound to Ω-terms). Suppose this goal is Ω-satisfiable according to either condition (1) or (3) given in the definition above. Then, X gets bound to $(\Omega(x{:}R(u,\hat{Y}),Q(\hat{Z},l),P[u,[\hat{A}|l],x))$. As in Case 1(a) every free non-input variable should be Ω-bound similarly.

Case 2: There is no non-input free variable in the arguments of $P(t_1,\ldots,t_n)$.

Case 2(a): None of the arguments of P is Ω-bound. This can happen only when all the arguments of P are either constant or skolem constant from the input variables. In this case no Ω-binding is generated since there are no free variables to be bound. However, P still denotes a condition which this particular instantiation of the input

variables must satisfy. Such conditions will become part of the *Premise* or *Lemma* as described in Section 3.3.

Case 2(b): Some arguments are Ω-bound. In this case an Ω-merging, as described in Case 1(b), needs to be performed, and all the Ω-bound variables should have their Ω-constraints changed accordingly. Once again we describe this process by an example. Suppose the Ω-unsatisfiable goal is $G(M, N)$ where $M \leftarrow \Omega(m{:}P(\hat{Y}, m))$ and $N \leftarrow [a \mid \Omega(n{:}Q(\hat{Y}, n))]$. In order to prevent this goal from failing, we rebind the values of M and N and add G as part of the new constraints. That is, M should be Ω-bound to $\Omega(m{:}P(\hat{Y}, m), Q(\hat{Y}, n), G(m, [a \mid n]))$ and N to $[a \mid \Omega(n{:}P(\hat{Y}, n), Q(\hat{Y}, m), G(m, [a \mid n]))$. Note that, in additon to the new constraint G, both of the Ω-constraints of the original bindings of M and N are now a part of the new Ω-constraints. Also note that $G(m, [a \mid n])$, instead of $G(m, n)$, is part of the new constraint since N was originally bound to $[a \mid \Omega(n{:}\cdots)]$.

We now apply this Ω-binding mechanism to the second subgoal, append($\hat{Y}$, Z, $L2$) of the 'append' example. append($\hat{Y}$, Z, $L2$) is not Ω-satisfiable since $\hat{Y}$ can unify with neither [ ] nor $[A \mid L]$. When invoking the Ω-binding procedure, Z is automatically hatted and becomes $\hat{Z}$. It is clear that Case 1(a) applies here since $L2$ is a free variable. Therefore, to 'satisfy' this goal, we assign $\Omega(l_2{:}\text{append}(\hat{Y}, \hat{Z}, l_2))$ to $L2$.

3.3. Premise and Lemma generations

Both premise and lemma are lists of constraints arising out of suspension of subgoals while the goals in 'prop' are Ω-satisfied. Note that suspended goals are converted into constraints via the Ω-binding mechanism. Every constraint so generated will end up exclusively as a part of either the premise or the lemma.

A premise is constructed by collecting all the (sub)goals that fall under Case 2(a) during the Ω-binding generation while the antecedents are being Ω-satisfied. Note that a goal will fall under Case 2(a) only when every variable in it is already skolemized. Hence, a premise represents conditions to be assumed on the skolemization of the input variables. An empty premise is considered to be true.

A lemma is intended to denote the formula on which the validity of the proposition being proved depends. Hence it is constructed after Ω-satisfying the consequent. The outcome of Ω-satisfying the consequent can either be produced through Ω-binding generation or without. In the latter case, the lemma constructed is the literal **true** because this means that the consequent can be satisfied regardless of the constraints upon which the validity of the antecedents depended. In the other case (when Ω-binding is employed), the consequent should have fallen under Case 2(a) or 2(b) during Ω-binding generation. Then the lemma is either an instance of the consequent (produced by Case 2(a)), or the Ω-constraint of an Ω-bound variable of the consequent subgoal (produced by Case 2(b)). Note that since the Ω-constraints produced from Case 2(b) are the same for all the Ω-bound variables (the only difference being the Ω-variables), the lemma produced is unique.

Getting back to the 'append' example, there are two instantiations for the variables (X, Y, Z), namely $([\ ], \hat{Y}, \hat{Z})$ and $([\hat{A}|\hat{X}], \hat{Y}, \hat{Z})$. Neither of them produces any premise (i.e., every subgoal from the antecedents can be satisfied by Ω-binding). So what remains is to satisfy the consequent, $L2 = L4$.

In the first case determined by the instantiation $([\ ], \hat{Y}, \hat{Z})$, $L2$ and $L4$ are bound to, respectively,

$$\Omega(l_2{:}\text{append}(\hat{Y}, \hat{Z}, l_2)) \quad \text{and} \quad \Omega(l_4{:}\text{append}(\hat{Y}, \hat{Z}, l_4)).$$

Since $\Omega(l_2{:}\text{append}(\hat{Y}, \hat{Z}, l_2) =_\Omega \Omega(l_4{:}\text{append}(\hat{Y}, \hat{Z}, l_4))$, the goal $L2 = L4$ is Ω-satisfied. Therefore, the lemma is **true**.

In the second instantiation $([\hat{A}|\hat{X}], \hat{Y}, \hat{Z})$, $L2$ and $L4$ are bound to, respectively,

$$[\hat{A}|\Omega(l_2{:}\text{append}(\hat{X}, \hat{Y}, l_1), \text{append}(l_1, \hat{Z}, l_2))]$$

and

$$[\hat{A}|\Omega(l_4{:}\text{append}(\hat{Y}, \hat{Z}, l_3), \text{append}(\hat{X}, l_3, l_4))].$$

The consequent $L2 = L4$ is not Ω-satisfiable since

$$\Omega(l_2{:}\text{append}(\hat{X}, \hat{Y}, l_1), \text{append}(l_1, \hat{Z}, l_2)) \neq_\Omega \Omega(l_4{:}\text{append}(\hat{Y}, \hat{Z}, l_3), \text{append}(\hat{X}, l_3, l_4)).$$

Therefore, Case 2(b) applies and the bindings of $L2$ and $L4$ now become

$$L_2 \leftarrow [\hat{A}|\Omega(l_2{:}\text{append}(\hat{X}, \hat{Y}, l_1), \text{append}(l_1, \hat{Z}, l_2), \text{append}(\hat{Y}, \hat{Z}, l_3), \text{append}(\hat{X}, l_3, l_4), l_2 = l_4)]$$

$$L_4 \leftarrow [\hat{A}|\Omega(l_4{:}\text{append}(\hat{X}, \hat{Y}, l_1), \text{append}(l_1, \hat{Z}, l_2), \text{append}(\hat{Y}, \hat{Z}, l_3), \text{append}(\hat{X}, l_3, l_4), l_2 = l_4)].$$

The lemma should, therefore, be

$$(\text{append}(\hat{X}, \hat{Y}, L1) \wedge \text{append}(L1, \hat{Z}, L2) \wedge \text{append}(\hat{Y}, \hat{Z}, L3) \wedge \text{append}(\hat{X}, L3, L4)) \supset L2 = L4,$$

produced from the Ω-constraint of either $L2$ or $L4$.

3.4. Limited forward chaining

The only purpose of using forward chaining in our method is to produce the possible induction hypotheses of the original proposition. Therefore, the forward chaining facility in our prover is very restricted. The idea is the following: Suppose the *Premise* and the *Lemma* for some instantiation $\bar{X}_i$ are $\{C_1, \ldots, C_k\}$ and $\{D_1, \ldots, D_m, Q\}$, where Q is an instance of the original consequent.[4] The set $\{C_1, \ldots, C_k, D_1, \ldots, D_m, Q\}$ is checked against the body of the PROLOG clause which defines the orignal proposition ('prop' in the 'append' example). If it is an

[4] It is easy to see, from the construction of *Lemma*, that an instance of the consequent must be in the lemma if the lemma is not the literal **true**.

instantiation of a superset of the clause body, then prop($\bar{X}_i$) is returned as the *Lemma.* (The *Premise* remains the same.) For instance in the second instantiation, ($[\hat{A}|\hat{X}]$, $\hat{Y}$, $\hat{Z}$), of the 'append' example, the lemma

$$\{(\text{append}(\hat{X}, \hat{Y}, L1), \text{append}(L1, \hat{Z}, L2), \text{append}(\hat{Y}, \hat{Z}, L3), \text{append}(\hat{X}, L3, L4), L2 = L4\}$$

matches with the body of the clause for prop(X, Y, Z) with X bound to $\hat{X}$, Y to $\hat{Y}$, and Z to $\hat{Z}$. Therefore, prop($\hat{X}$, $\hat{Y}$, $\hat{Z}$) is returned as the new *Lemma*, replacing the previous one. Since this lemma prop($\hat{X}$, $\hat{Y}$, $\hat{Z}$) is produced when trying to satisfy the goal prop($[\hat{A}|\hat{X}]$, $\hat{Y}$, $\hat{Z}$), it is the induction hypothesis in the obvious structural induction of lists.

3.5. Generating a well-spanned set of instantiations

The concepts described above will guarantee that all the initial subgoals (the goals in the clause body of prop(X, Y, Z)) will be successfully processed and *one* instantiation will be generated for each of the input variables. However, our goal is to show that the proposition is true for *all* possible instances of the input variables. Therefore we need a mechanism to find more instantiations (and reprocess the initial set of subgoals) until the domains of the input variables are completely covered. This is done by backtracking to selected choice points.

First we call a choice point (i.e., the point where the PROLOG execution does an *or*-split) a *marked choice point* if either

(1) some of the input variables are instantiated (to a nonvariable term) when matching the clause head or one of the subgoals in the clause body, or

(2) one of the goals of the clause body becomes a premise. In other words, a marked choice point is a backtracking point at which a different choice of the clause to match may result in different instantiations (skolemizations) for the input variables.

As mentioned before, an instantiation along with a premise and lemma is generated when we finish processing all the initial subgoals in the proposition. Then our method of generating another instantiation is to force a failure at this point and re-evaluate the input predicate ind_prove again. Our *failure forcing* mechanism is similar to the one in PROLOG for generating a second solution. The difference, however, is that in our case the theorem prover backtracks to the last *marked* choice point, but not to the last choice point as in PROLOG. The well-spannedness of the set of instantiations thus generated can be checked effectively [17, 27]. The well-spannedness property is guaranteed in the case of abstract data types if the operations which the predicates represent are totally defined.

4. An algorithmic description

The algorithm below shows the generation of a single instantiation for the input (i.e., the universally quantified) variables in the proposition, a *Lemma* and a *Premise*

for that particular instantiation. To generate a well-spanned set of instantiations, it is necessary to backtrack, as described before, the algorithm to the latest marked choice point, and re-execute the algorithm by choosing a different clause/fact to unify a (sub)goal.

We have used an algorithmic notation (rather than a PROLOG notation) to keep it free of PROLOG idiosyncrasies. The text within braces is intended to be treated as comments. The symbol ← should be treated as an operation that binds the value of the expression on the right-hand side to the variable on the left-hand side.

Ind_Prove(Prop($\bar{X}$), *Premise*, *Lemma*)
  Body ← body of the clause that unifies with Prop($\bar{X}$)
  Antecedents ← all but the last element of *Body*
  Consequent ← the last element of *Body*
  Premise ← empty list
  Lemma ← empty list
  Processgoals(*Antecedents*, *Premise*)
  Processconseq(*Consequent*, *Lemma*)
end {Ind_Prove}.

Processgoals(*Goals*, *Premise*)
  {*Goals* is a list of goals to be processed.}
  {*Premise* is a list of constraints on input variables.}
  {Processgoals processes every goal in *Goals*, skolemizing the input variables when necessary, and generating constraints or Ω-bindings when a goal is not satisfiable.}
  {The constraints generated are also asserted in the data base.}
  if not empty(*Goals*) **then**
    G ← first(*Goals*)
    Goals ← rest(*Goals*)
    case *G is*
      a successfully evaluable built-in predicate:
        Evaluate *G*
        Skolemize all free input variables in *G*
        Processgoals(*Goals*, *Premise*)
      not unifiable with any of the clause heads:
        Skolemize all free input variables in *G*
        Generate_Ω_binding(*G*, *Premise*)
        Processgoals(*Goals*, *Premise*)
        {Note: Generation of premises and their assertion into the data base are done inside Generate_Ω_bindings.}
      unifiable with a head provided a skolemized variable in G can be instantiated:
        Take the same action as in the previous case
      unifiable with a fact:

```
        Unify G
        Skolemize all free input variables in G
        Processgoals(Goals, Premise)
      otherwise:
        Unify G with the clause head
        Skolemize all variables that appear in terms that got
            bound (due to unification) to the input variables in G
        Goals ← append(body of the clause G was unified with, Goals)
        Processgoals(Goals, Premise)
    end case
  fi
end{Processgoals}

Processconseq(Consequent, Lemma)
  if Consequent is Ω_ satisfiable without altering the binding
    then Lemma ← true
    else
      Generate_Ω_bindings(Consequent, Lemma)
      Add to Lemma the Ω-constraint of an Ω-binding (if any) in
                                                        Consequent
  fi
end {Processconseq}.

Generate_Ω_bindings(G(t1, ..., tn), Constraints)
  case G(t1, ..., tn) is such that
    at least one of its arguments is a free non-input variable:
    case G(t1, ..., tn) is such that
      none of its arguments is Ω-bound:
        Bind every free non-input variable in G(t1, ..., tn)
                              to an Ω-term generated as described before
      some of its arguments are Ω-bound:
        Modify all Ω-bindings in the goal by performing Ω-merging
                                                    as described before
    end case
    there is no free non-input variable in its arguments:
    case G(t1, ..., tn) is such that
      none of its arguments is Ω-bound:
        Add G(t1, ..., tn) to the list currently bound to Constraints
        Assert G(t1, ..., tn) as a new fact in the data base
      some of its arguments are Ω-bound:
        Modify all Ω-bindings in the goal by performing Ω-merging
                                  and adding the goal to the Ω-constraint
    end case
```

end case
end {Generate_Ω_bindings}.

5. Discussion

5.1. *Interpreting the outcome of* **Ind_Prove**

Let $\bar{X}_i$ be an instantiation returned by the theorem prover, with the corresponding *Premise* $\text{Prem}(\bar{X}_i): A_1, \ldots, A_k$ and *Lemma* $\text{Lem}(\bar{X}_i): B_1, \ldots, B_l, Q$, where Q is (an instance of) the original consequent. Then the logical meaning of the outcome is

$$(A_1(\bar{X}_i) \wedge \cdots \wedge A_k(\bar{X}_i)) \supset (B_1(\bar{X}_i, \bar{Z}) \wedge \cdots \wedge B_l(\bar{X}_i, \bar{Z}) \supset Q(\bar{X}_i, \bar{Z}))$$

where all the free variables in the predicates are universally quantified.

One of the following situations is applicable to every set of lemma (Lem), premise (Prem), and instantiation ($\bar{X}_i$) returned by ind_prove.

(1) Lem is the literal **true** or Prem $\supset$ Lem.
(2) Lem has the form of $\text{prop}(\bar{X}_i')$ such that $\bar{X}_i' < \bar{X}_i$ under some well-founded ordering $<$ on terms.
(3) Lem is a false sentence.
(4) Lem is a proposition which does not fall into any of the above cases.

If every set of output returned by ind_prove in proving $\text{prop}(\bar{X})$ falls into either case (1) or case (2), then $\text{prop}(\bar{X})$ is true for all $\bar{X}$. Note that every instance of case (2) constitutes an induction step in the proof. Cases (1) and (2) are handled automatically by the procedure ind_prove.

Case (4) deals with the situation when the lemma generated from an instantiation is neither (obviously) true nor in the form of some induction hypothesis. In this case we can formulate a new proposition, and try to prove its correctness by invoking ind_prove on the new proposition. The new proposition to be proved can be constructed in a standard way from the lemma and premise as follows:

$$\text{newprop}(\bar{Y}) :- \text{Prem}(\bar{Y}), \text{Lem}(Y, \bar{Z}).$$

Note that the input variables for 'newprop' should include all the free variables in $\bar{X}_i$, which may be different from those in $\bar{X}$. (Note that all the hatted variables in $\bar{X}_i$ should be included in the list $\bar{Y}$ in 'newprop'.) Also, the consequent subgoal of 'newprop' is the last subgoal of $\text{Lem}(\bar{X}_i, \bar{Z})$.

Currently, in our system the construction of the new proposition is done manually. This process can be automated by treating the theorem prover as a problem reduction theorem prover and treat each invocation of ind_prove as one level of the problem reduction mechanism. In other words, for each level, ind_prove reduces the present proposition to (a set of) lemmas, which in turn become new propositions for the next level. This can be done indefinitely until all lemmas are either obviously true, false, or are inductive hypotheses of some kind. In order to reduce the number of levels needed, we may incorporate some 'linear' substrategies, such as the Boolean

reduction strategy in [7], for quickly checking tautologies. Note that if every instance of case (4) is proved successfully and the only other output situations fall under cases (1) and (2), then prop($\bar{X}$) is true for all $\bar{X}$.

Case (3) corresponds to the situation in which the lemma returned is false. In general, in such a situation we cannot come to any conclusion about the truth or falsity of prop($\bar{X}_i$). We have found, however, that in several examples it is possible to interpret in such a situation the outcome in a more useful way. More work needs to be done in this area.

5.2. A remark on Ω-satisfiability

In PROLOG, a goal is satisfied if it can be deduced to nothing but facts in the data base. The notion of Ω-satisfiability is different from PROLOG satisfiability in that we have extended PROLOG equality between terms to Ω-equivalence. In order to construct more effective lemmas, we also include the list of premises in the data base as PROLOG facts (as shown in the algorithm in Section 4). This will not change the soundness of the method since

(1) the premises do not contain any variables; they contain only constants and skolem constants (hatted variables); therefore, no new bindings will be created from using these premises;

(2) by the logical meaning of the lemma generated, the condition *Premise* $\supset$ *Lemma* needs to be verified for the proposition to be true; since the premise is already part of the antecedent of the condition, using them as assertions (or say PROLOG facts) for generating the lemma does not have the effect of adding new axioms.

5.3. More on generating lemmas

As mentioned before, the marked choice points are used while backtracking for finding new instantiations of the input variables. The choice points that are left unmarked, however, are not used in our current system. Backtracking through some of these unmarked choice points (those that occur after the last marked choice point) may result in different lemmas for the same instantiation. Although this feature is not yet in our system, it is not hard to incorporate it and produce a lemma which is a disjunction of all the lemmas produced from these unmarked choice points. The lemma so produced is weaker than the one from our original system (and thus presumably easier to verify). The disadvantage is that it may no longer be in Horn clause form.

5.4. Comparison

Proving inductive properties is considered one of the most difficult problems in automated theorem proving. To the authors' knowledge, two of the more successful methods that deal with this problem are the Boyer-Moore method [1] and the inductive term rewriting method [6, 10, 21].

A major difference between our method and that of Boyer-Moore is that our method is PROLOG-based as opposed to being LISP-based. Another significant difference is that we use unification on clause heads to find proper instantiations, as opposed to an artificial split of *nil* and *cons*. Therefore our method can be applied naturally to data types other than lists (see [9] for examples). Nor do we explicitly use an EVAL operator. Our methods of generating lemmas (as well as the separation of *Premise* and *Lemma*) are also different. On the other hand, the Boyer-Moore method has many powerful features, such as generalizations, which ours does not.

Recently, there has been some effort [13] to develop a verification system for proving PROLOG programs along the same lines that the Boyer-Moore system serves to verify LISP programs. Using techniques from natural deduction, [13] extend PROLOG's execution mechanism to handle an extended set of (non-Horn clause) formulas. In their system, induction is not automated completely. To carry out an inductive proof one has set up formulas denoting the basis and the inductive cases manually. Their system can then be used to verify the formulas so constructed. Recently, [12] have proposed a method of automatically generating induction formulas that can be used in conjunction with [13].

The term rewriting method was first developed to solve equational problems in universal algebra [15]. In the mid-70s, the method was employed by the researchers in abstract data types and became a useful tool that linked the programming language community and the theorem proving community. (PROLOG is an example of another application of theorem proving to programming language design.) In recent years the term rewriting method has been extended to prove (equational) inductive properties of data types without using induction implicitly by building the inductive step into the Knuth-Bendix completion procedure. Although the inductive reasoning ability of the inductive rewriting method is less powerful than the Boyer-Moore method, it seems more efficient when it is applicable, and it also provides a uniform environment for program development as well as verification (e.g., OBJ [5]). The rewriting method has also been generalized to richer theories [11] and to first-order theories in general [8, 19] and, thus, is no longer restricted to proving just equations. However, despite these extensions and more recent developments in conditional term rewriting methods [14, 19, 24], the term rewriting approach still cannot handle conditional definitions of data types (or, non-unit equations in first-order theory with equality) completely.

The following is a simple example [2]:

$$\begin{aligned} &\textit{Define}: \quad f(x, y) = \textbf{if } x = y \textbf{ then } g(x, y) \\ &\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } g(x, x). \\ &\textit{Prove}: \quad f(x, y) = g(x, x). \end{aligned}$$

The conditional rewriting methods of Remy and others (e.g., [19]) cannot prove the above statement even if the domain of x and y has a canonical term rewriting system (Remy's method gives a complete solution to the ground case).

Our inductive theorem proving method, being based on PROLOG, can also be used as a program environment for program development and verification [9]. Compared to the rewriting appraoch, our method can handle conditions more effectively since it is not equation-based. We illustrate this point by solving the above example using our method: We use F and G for the relations corresponding to f and g. The program for F is:

$$F(X, Y, Z) :- X = Y, G(X, Y, Z).^5$$
$$F(X, Y, Z) :- X \neq Y, G(X, X, Z).$$

and the proposition to be proved is:

$$\text{prop}(X, Y) :- F(X, Y, Z1), G(X, X, Z2), Z1 = Z2.$$

The prover yields

Instantiation	*Premise*	*Lemma*
(X, X)	**true**	**true**
(X, Y)	$X \neq Y$	**true**

Since the instantiations well-span the domain of X and Y, the proposition is correct.

Appendix A. Examples

A.1. Illustration of the append *example*

In the following we show how the theorem proving method produces a lemma and a premise for one of the instantiations, $([\hat{A} | \hat{X}], \hat{Y}, \hat{Z})$, of the input variables by providing a step by step illustration of the method on the proof of the associativity property of 'append' for the inductive case. The other instantiation, $([\ ], \hat{Y}, \hat{Z})$, is considerably easier to handle and will not be done here. We illustrate the processing of every goal that is considered during the proof process with one step corresponding to a single goal. At each step we indicate (as appropriate) the list of goals (the first of which is the current goal) remaining to be satisfied, a brief description of the action taken, the case applicable (Section 3.3) within the Ω-binding generation conditions and any change that occurs in the bindings of the variables.

(1) $\text{append}([\ ], L, L).$
(2) $\text{append}([A | L1], L2, [A | L3]) :- \text{append}(L1, L2, L3).$

$$\text{prop}(X, Y, Z) :- \text{append}(X, Y, L1), \text{append}(L1, Z, L2)$$
$$\text{append}(Y, Z, L3), \text{append}(X, L3, L4), L2 = L4.$$

Initialization

$$\textit{Antecedents} \leftarrow [\text{append}(X, Y, L1), \text{append}(L1, Z, L2),$$
$$\text{append}(Y, Z, L3), \text{append}(X, L3, L4)]$$
$$\textit{Consequent} \leftarrow L2 = L4.$$

[5] This program is more complicated than what a PROLOG programmer will actually do.

Processing of the antecedents
Goals: [append(X, Y, $L1$), append($L1$, Z, $L2$), append(Y, Z, $L3$), append(X, $L3$, $L4$)]
How the current goal is processed: unifies with clause (2)
Bindings of variables:

$$X \leftarrow [\hat{A} \mid \hat{X}], \qquad L1 \leftarrow [\hat{A} \mid L5]$$

Goals: [append($\hat{X}$, Y, $L5$), append($L1$, Z, $L2$), append(Y, Z, $L3$), append(X, $L3$, $L4$)]
How the current goal is processed: fails to unify
Case applicable for Ω-binding generation: 1(a)
Bindings of variables

$$Y \leftarrow \hat{Y}, \qquad L5 \leftarrow \Omega(l5{:}\text{append}(\hat{X}, \hat{Y}, l5))$$

Goals: [append($L1$, Z, $L2$), append(Y, Z, $L3$), append(X, $L3$, $L4$)]
How the current goal is processed: unifies with clause (2)
Bindings of variables:

$$L2 \leftarrow [\hat{A} \mid L6]$$

Goals: [append($L5$, Z, $L6$), append(Y, Z, $L3$), append(X, $L3$, $L4$)]
How the current goal is processed: fails to unify
Case applicable for Ω-binding generation: 1(b)
Bindings of variables:

$$Z \leftarrow \hat{Z}, \qquad L5 \leftarrow \Omega(l5{:}\text{append}(\hat{X}, \hat{Y}, l5), \text{append}(l5, \hat{Z}, l6)),$$

$$L6 \leftarrow \Omega(l6{:}\text{append}(\hat{X}, \hat{Y}, l5), \text{append}(l5, \hat{Z}, l6))$$

Goals: [append (Y, Z, $L3$), append(X, $L3$, $L4$)]
How the current goal is processed: fails to unify
Case applicable for Ω-binding generation: 1(a)
Bindings of variables:

$$L3 \leftarrow \Omega(l3{:}\text{append}(\hat{Y}, \hat{Z}, l3))$$

Goals: [append(X, $L3$, $L4$)]
How the current goal is processed: unifies with clause (2)
Bindings of variables:

$$L4 \leftarrow [\hat{A} \mid L7]$$

Goals: [append($\hat{X}$, $L3$, $L7$)]

How the current goal is processed: fails to unify
Case applicable for Ω-binding generation: 1(b)
Bindings of variables:

$$L7 \leftarrow \Omega(l7{:}\text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{X}, l3, l7))$$
$$L3 \leftarrow \Omega(l3{:}\text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{X}, l3, l7)).$$

Processing the consequent
Goals: $[L2 = L4]$
How the current goal is processed: not Ω-satisfiable
Case applicable for Ω-binding generation: 2(b)
Bindings of variables:

$$L7 \leftarrow \Omega(l7{:}\text{append}(\hat{X}, \hat{Y}, l5), \text{append}(l5, \hat{Z}, l6), \text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{X}, l3, l7), l6 = l7)$$
$$L6 \leftarrow \Omega(l6{:}\text{append}(\hat{X}, \hat{Y}, l5), \text{append}(l5, \hat{Z}, l6), \text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{X}, l3, l7), l6 = l7)$$

Premise $\leftarrow$ **true**
$$\textit{Lemma} \leftarrow (\text{append}(\hat{X}, \hat{Y}, l5), \text{append}(l5, \hat{Z}, l6), \text{append}(\hat{Y}, \hat{Z}, l3), \text{append}(\hat{X}, l3, l7), l6 = l7)$$
$$\leftarrow \text{prop}(\hat{X}, \hat{Y}, \hat{Z}).$$

A.2. A tree example

In this example we present two versions of the membership relation of a tree ('isin'). The first one treats a tree as an abstract object constructed from two constructors, 'emptytree" and 'mktree". The second one treats a tree as a flat list. 'isin' takes a tree and an element as arguments and determines whether the element is in the tree.

The abstract tree

isin(emptytree, E, **false**).
isin(tree(L, N, R), N, **true**),
isin(tree(L, N, R), E, B) :- $N \neq E$, isin(L, E, $B1$), isin(R, E, $B2$), $B = \text{or}(B1, B2)$.

The list tree

EMPTYTREE([ ], [ ]).
MKTREE([ ], N, R, [N | R]).
MKTREE([X | L], N, R, [X | T]) :- MKTREE(L, N, R, T).
ISIN([ ], X, **false**).
ISIN([X | L], X, **true**).
ISIN([X | L], Y, B) :- $X \neq Y$, ISIN(L, Y, B).

The first tree can be considered as an *abstraction* of 'isin' in the data type *Tree*, and the second one can be considered as an *implementation*. Both of them are executable PROLOG programs, nevertheless.

We want to show that the ISIN described in the second tree is the same as the 'isin' in the first tree. To put it informally, we want to prove that

if isin (tree(L, N, R), E, $B1$) **and**
MKTREE(L, N, R, T) **and** ISIN(T, E, $B2$),
then $B1 = B2$.

First note that the above description does indeed fit into the Horn clause formalism required for the propositions to be proved. Since MKTREE is defined in three different clauses, one proposition needs to be established for each. They are

prop0(T, E) :– EMPTYTREE([ ], T), ISIN(T, E, B), B = **false**.
prop1(L, N, R, N) :– MKTREE(L, N, R, T), ISIN(T, N, B), B = **true**.
prop2(L, N, R, E) :– $N \neq E$, ISIN(L, E, $B1$), ISIN(R, E, $B2$),
B = or($B1$, $B2$),
MKTREE(L, N, R, T), ISIN(T, C),
$B = C$.

The more interesting case is prop2, and the set of instantiations, with premises and lemmas, generated by ind_prove is the following:

Instantiations	*Premise*	*Lemma*
$L \leftarrow [\ ], N \leftarrow \hat{N}, E \leftarrow \hat{E}$		
(a) $R \leftarrow [\ ]$	$\hat{N} \neq \hat{E}$	**true**
(b) $R \leftarrow [\hat{E} \mid \hat{R}]$	$\hat{N} \neq \hat{E}$	**true**
(c) $R \leftarrow [\hat{Y} \mid \hat{R}]$	$\hat{N} \neq \hat{E}$, $\hat{Y} \neq \hat{E}$	**true**
$L \leftarrow [\hat{E} \mid \hat{L}], N \leftarrow \hat{N}, E \leftarrow \hat{E}$		
(a) $R \leftarrow [\ ]$	$\hat{N} \neq \hat{E}$	**true**
(b) $R \leftarrow [\hat{E} \mid \hat{R}]$	$\hat{N} \neq \hat{E}$	**true**
(c) $R \leftarrow [\hat{Y} \mid \hat{R}]$	$\hat{N} \neq \hat{E}$	**true**
$L \leftarrow [\hat{X} \mid \hat{L}], N \leftarrow, \hat{N}, E \leftarrow \hat{E}$		
(a) $R \leftarrow [\ ]$	**true**	prop2($\hat{L}$, $\hat{N}$, [ ], $\hat{E}$)
(b) $R \leftarrow [\hat{Y} \mid \hat{R}]$	$\hat{X} \neq \hat{E}$	prop2($\hat{L}$, $\hat{N}$, $[\hat{Y} \mid \hat{R}]$, $\hat{E}$)
(c) $R \leftarrow [\hat{E} \mid \hat{R}]$	$\hat{X} \neq \hat{E}$, $\hat{N} \neq \hat{E}$	TREE($\hat{L}$, $\hat{N}$, $[\hat{E} \mid \hat{R}]$, L), ISIN(L, $\hat{E}$, B), B = **true**

The last instantiation has an unresolvable lemma. As described before, the lemma can be converted into a new proposition which will be further proved by the same method. In this particular problem, the new proposition is:

newprop(L, N, R, E) :– $N \neq E$, TREE(L, N, $[E \mid R]$, M),
ISIN(M, E, B), B = **true**.

The proof generated by ind_prove for it is

Instantiations	*Premise*	*Lemma*
(1) newprop([ ], $\hat{N}$, $\hat{R}$, $\hat{E}$)	**true**	**true**
(2) newprop($[\hat{X} \mid \hat{L}]$, $\hat{N}$, $\hat{R}$, $\hat{E}$)	**true**	newprop($\hat{L}$, $\hat{N}$, $\hat{R}$, $\hat{E}$)

Another interesting characteristic of our method can be seen from this example. Note that the prover partitions, automatically, the domain into *nine* parts:

prop2([ ], N, [ ], E)	prop2([ ], N, $[E \mid R]$, E)	prop2([ ], N, $[Y \mid R]$, E)
prop2($[E \mid L]$, N, [ ], E)	prop2($[E \mid L]$, N, $[E \mid R]$, E)	prop2($[E \mid L]$, N, $[Y \mid R]$, E)
prop2($[X \mid L]$, N, [ ], E)	prop2($[X \mid L]$, N, $[E \mid R]$, E)	prop2($[X \mid L]$, N, $[Y \mid R]$, E)

instead of the usual three-part partition. This is because our prover partitions domains according to how the predicates are defined and not simply to the structure of the data type.

6. References

[1] R. Boyer and J.S. Moore, *A Computational Logic* (Academic Press, New York, 1979).

[2] D. Brand, J.A. Darringer and W.H. Joyner, Completeness of conditional reductions, in: *Proc. 4th Conf. on Automated Deduction* (1979) 36–42.

[3] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer, Berlin, 1981).

[4] S.K. Debray and D.S. Warren, Detection and optimization of functional computations in Prolog, in: *Proc. 3rd Internat. Conf. on Logic Programming*, London, U.K. (1986) 490–504.

[5] J.A. Goguen and J.J. Tardo, An introduction to OBJ: a language for writing and testing formal algebraic program specifications, in: *Proc. Conf. on Specification of Reliable Software*, Cambridge, MA (1979) 170–189.

[6] J.A. Goguen, How to prove algebraic inductive hypothesis without induction, in: *Proc. 5th Conf. on Automated Deduction* (1980) 356–372.

[7] J. Hsiang, Topics in automated theorem proving and program generation, Tech. Rept. UIUCDCS-R-82-1113, Dept. of Computer Science, Univ. of Illinois at Urbana Champaign, 1982.

[8] J. Hsiang, Refutational theorem proving using term rewriting systems, *Art. Intell.* **25** (1985) 255–300.

[9] J. Hsiang and M.K. Srivas, A PROLOG environment for developing and reasoning about data types, in: *Proc. Internat. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, Berlin (1985) 276–293.

[10] G. Huet and J.M. Hullot, Proofs by induction in equational theories with constructors, in: *Proc. 21st IEEE Symp. on Foundations of Computer Science* (1980) 797–821.

[11] J. Jouannaud and H. Kirchner, Completion of a set of rules modulo a set of equations, in: *Proc. 11th Symp. on Principles of Programming Languages*, Salt Lake City, UT (1984) 83–92.

[12] T. Kanamori and H. Fujita, Formulation of induction formulas in verification of Prolog programs, in: *Proc. 8th Internat. Conf. on Automated Deduction*, Oxford, U.K. (1986) 281–299.

[13] T. Kanamori and H. Seki, Verification of Prolog programs using an extension of execution, in: *Proc. Third Internat. Conf. on Logic Programming*, London, U.K., (1986) 475–489.

[14] S. Kaplan, Fair conditional term rewriting systems, in: *Proc. 12th ICALP*, Nafplion, Greece, (1985).

[15] D.E. Knuth and P.B. Bendix, Simple word problems in universal algebras, in: J. Leach, ed., *Computational Algebra* (Pergamon Press, Oxford, 1970) 263–297.

[16] W.A. Kornfeld, Equality in Prolog, in: *Proc. 8th IJCAI*, Karlsruhe, Germany (1983) 514–519.

[17] E. Kounalis, Completeness in data type specifications, in: *Proc. EUROCAL '85*, Lecture Notes in Computer Science **204** (Springer, Berlin, 1985) 348–362.

[18] D.S. Lankford, Canonical inference, Tech. Rept. ATP-32, Dept of Computer Science, Univ. of Texas at Austin, 1975.

[19] D.S. Lankford, Some new approaches to the theory and application of conditional term rewriting systems, Tech. Rept., Louisiana Tech. Univ., 1979.

[20] J.W. Lloyd and R.W. Topor, Making Prolog more expressive, *J. Logic Programm.* **1**(3) (1984) 225–240.

[21] D.R. Musser, On proving inductive properties of abstract data types, in: *Conf. Record of the Seventh Ann. ACM Symp. on Principles of Programming Languages*, Las Vegas, NV (1980) 154–162.

[22] D.A. Plaisted, The occur-check problem in Prolog, in: *Proc. 1984 Internat. Symp. on Logic Programming*, Atlantic City, NJ (1984) 272–280.

[23] R. Reiter, On closed world data bases, in: H. Gallaire and J. Minker, ed., *Logic and Data Bases* (Plenum Press, New York, 1978) 55–76.

[24] J.L. Remy, Conditional term rewriting system for abstract data types, Tech. Rept., University of Nancy, France, 1983, submitted for publication.

[25] M.K. Srivas, Automatic synthesis of implementations for abstract data types from algebraic specifications, MIT/LCS Tech. Rept. 276, Laboratory for Computer Science, MIT, 1982.

[26] M.E. Stickel, A Prolog technology theorem prover, in: *Proc. 1984 Internat. Symp. on Logic Programming*, Atlantic City, NJ (1984) 212–219.

[27] J. Thiel, Stop losing sleep over uncomplete data type specifications, in: *Proc. 11th Symp on Principles of Programming Languages*, Salt Lake City, UT, 1984.