Implicit induction in conditional theories

Adel Bouhoula, Michaël Rusinowitch

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implicit Induction in Conditional Theories

Adel BOUHOULA
Michaël RUSINOWITCH

N° 2045
Septembre 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport
de recherche

1993

Implicit Induction in Conditional Theories*[†][‡]

Adel Bouhoula and Michaël Rusinowitch
CRIN & INRIA-Lorraine
BP 239, 54506 Vandoeuvre-lès-Nancy, France
email: {bouhoula,rusi}@loria.fr

September 18, 1993

Abstract

We propose a new procedure for proof by induction in conditional theories where case analysis is simulated by term rewriting. This technique reduces considerably the number of variables of a conjecture to be considered for applying induction schemes (inductive positions). Our procedure is presented as a set of inference rules whose correctness has been formally proved. Moreover, when the axioms are ground convergent it is possible to apply the system for refuting conjectures. The procedure is even refutationally complete for conditional equations with boolean preconditions over free constructors (under the same hypotheses). The method is entirely implemented in the prover **SPIKE**. This system has proved interesting examples in a completely automatic way, that is, without interaction with the user and without ad-hoc heuristics. It has also proved the challenging Gilbreath card trick, with only 2 easy lemmas.

keywords: Theorem proving, Term rewriting systems, Implicit induction, Conditional theories.

Induction Implicite dans les théories Conditionnelles

Résumé

On propose une nouvelle procédure de preuve par induction dans les théories conditionnelles où l'analyse par cas est simulée par la réécriture. Cette technique réduit considérablement le nombre des variables d'une conjecture à considérer pour l'application des schémas d'induction (positions inductives). Notre procédure est présentée par un ensemble de règles d'inférence dont la correction a été formellement prouvée. De plus, si les axiomes sont convergents sur les termes clos alors il est possible d'appliquer le système pour réfuter des conjectures non valides. La procédure est aussi réfutationnellement complète pour les équations conditionnelles avec préconditions booléennes sur des constructeurs libres (sous les mêmes hypothèses). Notre méthode est entièrement implémentée dans le prouveur **SPIKE**. Ce système a prouvé des exemples intéressants d'une manière automatique, i.e. sans interaction avec l'utilisateur et sans heuristiques. Par exemple, 2 lemmes suffisent pour prouver le tour de cartes de Gilbreath.

mots clés: Preuve automatique, Système de réécriture, Induction implicite, Théorie conditionnelles.

[*] This is an extended version of a paper presented at the International Joint Conference on Artificial Intelligence, Chambéry (France), 1993.

[†] We thank our colleagues from the IndusMind group for stimulating discussions.

[‡] Partly supported by the PRC mécanisation du raisonnement and the Esprit BRA workshop COMPASS

1 Introduction

Formal methods are more and more frequently adopted by industry for hardware and software verification. They require efficient automatic tools to relieve designers and programmers of the related proof obligations. Mathematical induction is essential as a technique for building formal proofs in this context. Its power is expressed by the successes of Nqthm [Boyer and Moore, 1979] that has been for many years the only significant automated theorem proving system for induction. However Nqthm requires a lot of interaction with the user. For instance many lemmas need to be given to Nqthm as milestones even for simple proof tasks.

Another direction for automating induction was proposed in the early eighties, the *inductionless induction* technique [Musser, 1980; Huet and Hullot, 1982; Kapur and Musser, 1987] whose principle is to simulate induction by term rewriting. This method is refutational and does not require human interaction. While very limited at the beginning, its domain of application has widened considerably, thanks to the contributions of [Jouannaud and Kounalis, 1986] who relaxed the conditions on *constructor symbols* and of [Fribourg, 1986] who showed that only linear derivations were needed (see also [Bachmair, 1988]). More recently the method has been completely freed from the completion framework [Kounalis and Rusinowitch, 1990a; Reddy, 1990]. It has now become possible to apply it to conditional equational theories [Kounalis and Rusinowitch, 1990b; Bouhoula *et al.*, 1992a]. Inductionless induction in our new setting reduces to *firstly* instanciating conjectures by induction schemes called *test sets* and *secondly* simplifying them by axioms, other conjectures or induction hypotheses. Every iteration generates new lemmas that are processed in the same way as the initial conjectures. The method does not require any hierarchy between the lemmas. They are all stored in a list and using conjectures for mutual simplification simulates *simultaneous induction.* The system SPIKE has been developed [Bouhoula *et al.*, 1992b] on this principle and incorporates many optimizations such as powerful simplification techniques. To our knowledge, this system is the only one that can prove and disprove inductive theorems in conditional theories without any interaction.

However computer experiments have convinced us of the necessity of introducing a proper rule to perform case reasoning. Case analysis is a fundamental reasoning technique. A typical instance of it is the cut rule that consists in splitting a goal formula A along another formula C for generating two subgoals $C \Rightarrow A$ and $\neg C \Rightarrow A$. The main difficulty, already recognized by logicians a long time ago, relies in the choice of the cut formula C. A natural solution when dealing with theories axiomatized by Horn clauses (or conditional equations) is to use the negative literals of the axioms as cut formulas. This approach is frequently used in the context of conditional rewrite system when a conditional rule $C \Rightarrow l \rightarrow r$ may reduce a goal A to subgoals $C \Rightarrow A[r/l]$ (i.e. term l is replaced by r in A) and $\neg C \Rightarrow A$.

The problem is now that one of the subgoals is not smaller than the initial goal and a lot of control is needed to avoid divergence of the process since a similar case analysis can be applied again to $\neg C \Rightarrow A$. This has motivated us to introduce a new case analysis rule that allows one to splitting a goal A into subgoals $C_i \Rightarrow A[l_i/r_i]$ and $\vee_i C_i$. Since in the context of conditional rewrite system all subgoals are strictly smaller than the initial goal the search space is much more controlled. Related approaches were proposed independently by [Bronsard and Reddy, 1990] and [Bevers, 1993]. However, since their inference systems are unable to handle non Horn clauses, they cannot prove the $\vee_i C_i$ formulas otherwise than by external means. On the contrary, our proof technique applies to non Horn clauses as well. The disjunction $\vee_i C_i$ is added to the other conjectures and does not require particular treatment. Therefore our setting is very homogeneous and permits one to extend our basic inference rules by various optimizations without loosing correctness and completeness. In particular we have a notion of inductive positions defining the subset of variables of a conjecture that can be instanciated by induction schemes and we have proved that these positions are the only ones needed for completeness. The restriction of induction to these positions reduces drastically the search space. The importance of such restrictions was recognized a long time ago by [Boyer and Moore, 1979].

The paper is organized as follows. In Section 2 we introduce the basic definitions about term rewriting. In Section 3 we define the notions of inductive theory and inductive rewriting, which is a fundamental tool for proving inductive theorems. We define in section 4 inductive positions and test sets. Section 5 presents our technique of simulating case reasoning by rewriting. This technique reduces considerably the number of inductive positions to be considered. The strategy can be embedded in a correct set of inference rules described in Section 6. When the axioms are ground convergent and the defined functions are completely defined then it is possible to apply the system for refuting conjectures (subsection 6.3). In Section 7, the strategy is even proved refutationally complete for conditional equations with boolean preconditions (under previous hypotheses). Some optimizations are given in section 8 and computer experiments with SPIKE are discussed in Section 9.

SPIKE has proved the challenging Gilbreath card trick. The proof is based on 5 (2, with H. Zhang's formulation) lemmas that are easy to understand. This example was treated by B. Boyer and H. Zhang. Unlike us, they require a lot of lemmas, some of them being non-obvious. In appendix, we give a detailed account of SPIKE's proof.

2 Basic concepts

We assume that the reader is familiar with the basic notions of rewrite systems. We introduce the essential terminology below and refer to [Dershowitz and Jouannaud, 1990] for more detailed presentations.

2.1 Terms and substitutions

A many-sorted signature Σ is a pair (S, F) where S is a set of *sorts* and F is a finite set of function symbols. We assume that we have a partition of F in two subsets, the first one, C, contains the *constructor symbols* and the second, D, is the set of *defined symbols.*

Let X be a family of free sorted variables and let $T(F, X)$ be the set of well-sorted F-terms. $Var(t)$ stands for the set of all variables appearing in t and $\sharp(x,t)$ denotes the number of occurrences of the variable x in t. A variable x in t is *linear* iff $\sharp(x,t) = 1$. If $Var(t)$ is empty then t is a *ground* term. By $T(F)$ we denote the set of all ground terms. From now on, we assume that there exists at least one ground term of each sort.

For any term t, $occ(t) \subseteq N^*$ denotes its set of positions and the expression t/u denotes the *subterm of t at a position u.* The root position is written ϵ. We write $t[s]_u$ (resp. $t[s]$) to indicate that s is the subterm of t at position u (resp. at some position). Also let $t(u)$ denote the symbol of t at position u. A position u in a term t is said to be *a strict position* if $t(u) = f \in F$, a *variable position* if $t(u) = x \in X$ and $\sharp(x,t) = 1$, a *non-linear variable position* if $t(u) = x \in X$ and $\sharp(x,t) > 1$. We use $sdom(t)$ to denote the set of strict positions in t. If u is a position, then $|u|$ (the *length* of the corresponding string) gives us its *depth.*

A Σ – *substitution* assigns Σ – *terms* of appropriate sorts to variables. Composition of substitutions σ and η is written by $\sigma\eta$. The Σ – *term* $t\eta$ obtained by applying a substitution η to t is called an *instance* of t. If η is a ground substitution (i.e. $x\eta$ is ground for every x), we say that $t\eta$ is a *ground instance* of t.

2.2 Conditional Equations and Clauses

Let $\Sigma = (S, F)$ be a signature. A Σ-*equation* is a pair $e = e'$ where e, $e' \in T(F, X)$ are terms of the same sort. A *conditional* Σ-equation is either a Σ-equation or an expression of one of the following forms: $(e_1 \wedge \ldots \wedge e_n \Rightarrow e)$, or $(e_1 \wedge \ldots \wedge e_n \Rightarrow)$, or $(\Rightarrow)$ where $e, e_1, \ldots e_n$ are Σ-equations. $e_1, \ldots e_n$ are the *conditions* and e is the *conclusion.* A Σ-clause is an expression of the form $\neg e_1 \ \vee \ \neg e_2 \ \vee \ \ldots \ \vee \ \neg e_n \ \vee \ e'_1 \ \vee \ \ldots \ \vee \ e'_m$. When Σ is clear from the context, we omit the prefix Σ. We identify a conditional equation and its corresponding representation as a

Horn clause. A clause is *positive* if $\neg$ does not occur in it. Let c_1 and c_2 two clauses such that $c_1\sigma$ is a subclause of c_2 for some substitution σ, then we say that c_1 subsumes c_2.

In this paper, axioms are built from conditional equations and goals to be proved are clauses, i.e. disjunction of equational literals, since $=$ is the only predicate [1]. The symbol $\equiv$ is used for syntactic equality between two objects.

2.3 Rewrite Relations

2.3.1 Preliminaries

Given a binary relation $\rightarrow$, $\rightarrow^*$ denotes its reflexive and transitive closure. Let a and b be two terms, we write $a \downarrow b$, if there exists c such that $a \rightarrow^*$ c and $b \rightarrow^*$ c. A relation $\rightarrow$ is noetherian if there is no infinite sequence $t_1 \rightarrow t_2 \rightarrow \ldots$. In the following, we suppose given a *reduction ordering* $\succ$ on the set of terms, that is a transitive irreflexive relation that is noetherian, monotonic ($s \succ t$ implies $w[s] \succ w[t]$) and stable ($s \succ t$ implies $s\sigma \succ t\sigma$). We also assume that the ordering $\succ$ can be extended consistently when adding new constants to the signature. The multiset extension of an ordering $\succ$ will be denoted by $\gg$. Given a congruence relation $\approx$ on terms that is stable ($s \approx t$ implies $s\sigma \approx t\sigma$) and compatible with $\succ$ ($s \succ t, s \approx s', t \approx t'$ implies $s' \succ t'$) we define $\succeq$ as $\succ \cup \approx$. We write $s \not\Join t$ when $s \not\succeq t$ and $t \not\succeq s$.

A conditional equation $a_1 = b_1 \wedge \ldots a_n = b_n \Rightarrow s = t$ will be written as $a_1 = b_1 \wedge \ldots \wedge a_n = b_n \Rightarrow s \rightarrow t$ if $\{s\sigma\} \gg \{t\sigma, a_1\sigma, b_1\sigma, \ldots, a_n\sigma, b_n\sigma\}$ for all ground substitutions σ; in that case we say that $a_1 = b_1 \wedge \ldots a_n = b_n \Rightarrow s \rightarrow t$ is a *conditional rule.* The term s is the *left-hand side* of the rule. A set of conditional rules is a *rewrite system.*

2.3.2 Conditional Rewriting

The idea of rewriting is to impose a direction when using equations in proofs. This direction is indicated by an arrow when it is independent from the instantiation: $l \rightarrow r$ means that we can replace l by r in any context. When an instance of a conditional equation is orientable and has a valid conditional part it can be applied as a rule. The conditions are checked recursively. Termination is ensured by requiring the conditions to be smaller (w.r.t. the reduction ordering $\succ$) than the conclusion.

Definition 2.1 (Conditional Rewriting) *Let R be a set of conditional equations. Let a be a term and u a position in a. We write: $a[s\sigma]_u \rightarrow_R a[t\sigma]_u$ if there is a substitution σ and a conditional equation $\wedge_{i=1}^n a_i = b_i \Rightarrow s = t$ in R such that:*

1. *$s\sigma \succ t\sigma$.*

2. *$\forall i \in [1..n]\ a_i\sigma \downarrow_R b_i\sigma$.*

3. *$\{a[s\sigma]_u\} \gg \{a_1\sigma,\ b_1\sigma,\ \ldots\ ,a_n\sigma,\ b_n\sigma\}$.*

A term t is reducible w.r.t. to $\rightarrow_R$ if there is a term t' such that $t \rightarrow_R t'$. Otherwise we say t is *R-irreducible.* The system R will be qualified as *ground convergent* if for all a, b in $T(F)$

$$R \models a = b \textit{ implies } a \downarrow_R b$$

Note that when R is a rewrite system the relation $\rightarrow_R$ is similar to the notion of decreasing rewriting of Dershowitz, Okada and Sivakumar [Dershowitz *et al.*, 1988].

[1] It is straightforward to extend the following results to signatures with other predicates than equality.

2.3.3 Sufficient completeness

When for all possible arguments the result of a defined operator can be expressed with constructors only, we say that this operator is completely defined w.r.t. the constructors. This requirement is very natural when building specifications in a structured way [Guttag, 1978]. Here is a more formal definition:

Definition 2.2 *Let R be a rewrite system, let C be a set of constructors and D be a set of defined operator. The operator $f \in D$ is completely defined w.r.t. C iff for all $t_1, \ldots, t_n$ in $T(C)$, there exists t in $T(C)$ such that $f(t_1, \ldots, t_n) \rightarrow^*_R t$.*

Although this property is in general undecidable, our system SPIKE offers facilities to check and complete definitions. The program builds a pattern tree for every defined operator f (see [Kounalis, 1990]). The leaves of the tree give a partition of the possible arguments for f. Then if all the leaves are reducible, the answer is affirmative. If one of the leaves is not reducible then SPIKE suggests a new rule for completing the specification. This rule is not entirely determined but rather a possible schema for it is proposed, namely: *(condition, left-hand-side)*. Once the user has chosen the new rule, usually by simply giving its right-hand side, SPIKE replays the test.

Example 2.1 *Consider the specification of lists of natural numbers with an "insert" operation and a "sorted" predicate on lists that is true iff a list is ordered. We have $C = \{0, s, True, False, cons, Nil\}$ and $D = \{\leq, sorted, insert\}$.*

$$\begin{array}{rcll}
True = False & \Rightarrow & & (1)\\
0 \leq x & \rightarrow & True & (2)\\
s(x) \leq 0 & \rightarrow & False & (3)\\
s(x) \leq s(y) & \rightarrow & x \leq y & (4)\\
sorted(Nil) & \rightarrow & True & (5)\\
sorted(cons(x, Nil)) & \rightarrow & True & (6)\\
x \leq y = False \Rightarrow sorted(cons(x, cons(y, z))) & \rightarrow & False & (7)\\
x \leq y = True \Rightarrow sorted(cons(x, cons(y, z))) & \rightarrow & sorted(cons(y, z)) & (8)\\
insert(x, Nil) & \rightarrow & cons(x, Nil) & (9)\\
x \leq y = True \Rightarrow insert(x, cons(y, z)) & \rightarrow & cons(x, cons(y, z)) & (10)\\
x \leq y = False \Rightarrow insert(x, cons(y, z)) & \rightarrow & cons(y, insert(x, z)) & (11)
\end{array}$$

All defined symbols are completely defined:

```
          ----------------------------------------------
          >>> Display of pattern tree of <=  <<<
          ----------------------------------------------


 x1<=x2
   0<=x2  ok
   S(x3)<=x2
     S(x3)<=0  ok
     S(x3)<=S(x1)  ok
```

```
          ----------------------------------------------
          >>> Display of pattern tree of sorted  <<<
          ----------------------------------------------

 sorted(x1)
   sorted(Nil)  ok
   sorted(Cons(x2,x3))
     sorted(Cons(x2,Nil))  ok
     sorted(Cons(x2,Cons(x1,x4)))  ok
       (x2<=x1)=False::False
       (x2<=x1)=True::sorted(Cons(x1,x4))

          ----------------------------------------------
          >>> Display of pattern tree of insert  <<<
          ----------------------------------------------

 insert(x1,x2)
   insert(x1,Nil)  ok
   insert(x1,Cons(x3,x4))  ok
     (x1<=x3)=True::Cons(x1,Cons(x3,x4))
     (x1<=x3)=False::Cons(x3,insert(x1,x4))

---> every defined symbol is completely defined.
```

3 Induction

3.1 Inductive theory

Given a set of Horn clauses Ax on the signature Σ, we recall that a Herbrand (or term-generated) model of Ax is a model of Ax whose domain is the set of ground terms (axioms for equality are implicitly assumed to be valid, too). A formula $\mathcal{F}$ is a *deductive theorem of* Ax if it is valid in any model of Ax. This will be denoted by $Ax \models \mathcal{F}$. Deductive theorems can be proved by refutation, by deriving a contradiction from $\neg\mathcal{F} \wedge Ax$. Usually $\neg\mathcal{F}$ is transformed into a universal sentence $\mathcal{U}$ by introducing *skolem functions.* Hence the signature Σ has to be extended.

The theory of Ax, which is the one we are interested in, is the class of sentences that are true in the minimal Herbrand (or initial) model of Ax. For detailed definitions of initial models see, for instance, [Padawitz, 1988]. Every element in the domain of a Herbrand model is denoted by a ground term built on the signature of Ax. But since ground terms can easily be well-ordered, induction is available as a natural technique to prove sentences in these models. This is why we shall call *inductive theory* of Ax the class of sentences that are valid in the minimal Herbrand model of Ax:

Definition 3.1 *Let Ax be a set of Horn clauses on the signature Σ. A clause e is an* **inductive consequence** *of Ax if it is valid in the minimal Herbrand model of Ax. This will be denoted by $Ax \models_{ind} e$.*

The following proposition gives us a useful characterization of inductive consequences:

Proposition 3.1 *A clause $\neg C_1 \vee \ldots \vee \neg C_n \vee P_1 \vee \ldots \vee P_m$ is an inductive consequence of Ax if and only if for any ground substitution σ,*

$$(for\ all\ i : Ax \models C_i\sigma)\ \ implies\ \ (there\ exists\ j\ such\ that\ Ax \models P_j\sigma)$$

For clauses validity in all Herbrand models differs, in general, from validity in the initial model. However, these two notions of validity coincide for unconditional equations [Padawitz, 1988].

3.2 Inductive rewriting

To simplify goals, we extend the conditional rewriting relation definition 2.1, so that we can check the conditions of a rule to be applied to a clause C with inductive hypothesis, other conjectures and the premisses of C, considered as an implication formula.

Let us first introduce a few notations. Let $C \equiv \neg(a_1 = b_1) \vee \ldots \vee \neg(a_n = b_n) \vee (c_1 = d_1) \vee \ldots \vee (c_m = d_m)$. Then we denote by $prem(C)$ the set of negated atoms of C: $\{a_i = b_i\}_{i=1,n}$. The expression $(a = b)^\varepsilon$ denotes the literal $a = b$ if $\varepsilon = +$ and the literal $\neg(a = b)$ if $\varepsilon = -$. The skolemized clause $\overline{C}$ of C is the clause obtained by substituting every variable of C by a new constant. We recall that $\succ$ can be extended consistently to terms with new symbols.

Definition 3.2 (Inductive rewriting) *Let R and W be two sets of conditional equations. Consider a clause $C \equiv (a = b)^\varepsilon \vee r$ and its skolemized version $\overline{C} \equiv (\overline{a} = \overline{b})^\varepsilon \vee \overline{r}$. We write: $a \mapsto_{R[W];r} a'$ if $a \succ a'$ and:*
either $\overline{a} \mapsto_{prem(\overline{r})} \overline{a'}$,
or *there exists a position u in a, a substitution σ and a conditional equation $\wedge_{i=1}^{n} a_i = b_i \Rightarrow s = t$ in R such that:*

1. $a \equiv a[s\sigma]_u$ *and* $a' \equiv a[t\sigma]_u$.

2. $\{a[s\sigma]_u\} \gg \{a_1\sigma,\ b_1\sigma,\ \ldots\ , a_n\sigma,\ b_n\sigma\}$.

3. $\forall i \in [1..n]\ \ \exists c'_i, d'_i$ *such that* $a_i\sigma \mapsto^*_{R\cup W\ [\ ];r} c'_i$ *and* $b_i\sigma \mapsto^*_{R\cup W\ [\ ];r} d'_i$ *and* $\overline{c'_i} =_{prem(\overline{r})} \overline{d'_i}$.

where $=_{prem(\overline{r})}$ is the congruence generated by $prem(\overline{r})$.

The set W in the definition is intended to contain induction hypotheses in the proof system described below. Inductive rewriting can be viewed as a generalization of both the rewriting relation defined in [Kounalis and Rusinowitch, 1990b] and contextual rewriting [Zhang, 1993].

Example 3.1 *With R as in example 2.1, here is an example of a clause which is inductively rewritten by its own conditions:*

```
Simplification of:
s(x2)<x3 = False =>
   x1<x2 = True, s(x1)<x3 = True, False = sorted(Cons(s(s(x2)),Cons(s(x3),Nil)))
 by R[ ];r:
    s(x2)<x3 = False => x1<x2 = True, s(x1)<x3 = True, False = False
```

In this example we have:

$$sorted(Cons(s(s(x2)), Cons(s(x3), Nil))) \mapsto_{R[\ ];\{s(x2)<x3=False\}} False$$

Example 3.2 *The following rules define odd and even for nonnegative integers:*

$$\begin{array}{rcll}
even(0) & \rightarrow & True & (12)\\
even(s(0)) & \rightarrow & False & (13)\\
even(s(s(x))) & \rightarrow & even(x) & (14)\\
even(x) = True \Rightarrow odd(x) & \rightarrow & False & (15)\\
even(x) = False \Rightarrow odd(x) & \rightarrow & True & (16)\\
True = False \Rightarrow & & & (17)
\end{array}$$

We consider the following conjectures (+ is defined as usual):

```
EO={odd(s(x+x))= True, odd(x+x)=False, even(s(x+x))=False, even(x+x)=True}
```

We show an example where a clause is inductively rewritten with a conjecture from the initial set E_0 (H0 the initial set of inductive hypothesis is empty):

```
Simplification of:
    odd(s(x1+x1)) = True by R[HO U EO]:
    True = True

Simplification of:
    odd(x1+x1) = False by R[HO U EO]:
    False = False
```

Remark that inductive rewriting possesses a kind of stability in the sense that for any term s, t, any clause r and substitution τ, $s \mapsto_{R[W];r} t$ implies $s\tau \mapsto_{R[W];r\tau} t\tau$. If W is empty, $\mapsto_{R[W];r}$ will be denoted by $\mapsto_{R;r}$.

Lemma 1 *For all substitutions τ: $s \mapsto_{R[W];r} t$ implies $s\tau \mapsto_{R[W];r\tau} t\tau$.*

proof: We have two cases:

1. $W = \emptyset$ then a proof is done by noetherian induction on s with $\succ$. We have $s \succ t$ then $s\tau \succ t\tau$. Now, we consider two cases:
   (a) $s \mapsto_{R;r} t$ with $\overline{s} \mapsto_{prem(\overline{r})} \overline{t}$, then $\overline{s\tau} \mapsto_{prem(\overline{r\tau})} \overline{t\tau}$.
   (b) There exists a conditional equation $\wedge_i\ a_i = b_i \Rightarrow l = r$ of R such that:
      i. $s/u = l\sigma$ for a position u in s and a substitution σ, then we have also: $s\tau/u = l\sigma\tau$.
      ii. $t = s[r\sigma]_u$, clearly: $t\tau = s\tau[r\sigma\tau]_u$.
      iii. $\{a\} \gg \{a_1\sigma, \ldots, b_n\sigma\}$ implies that $\{a\tau\} \gg \{a_1\sigma\tau, \ldots, b_n\sigma\tau\}$.
      iv. $\forall i \in [1 \ldots n]$:
         A. $a_i\sigma \mapsto^*_{R;r} c'_i$, since $s \succ a_i\sigma$, by the induction hypothesis: $a_i\sigma\tau \mapsto^*_{R;r\tau} c'_i\tau$.
         B. $b_i\sigma \mapsto^*_{R;r} d'_i$, since $s \succ b_i\sigma$, by the induction hypothesis: $b_i\sigma\tau \mapsto^*_{R;r\tau} d'_i\tau$.
         C. $\overline{c'_i} =_{prem(\overline{r})} \overline{d'_i}$ which implies: $\overline{c'_i\tau} =_{prem(\overline{r\tau})} \overline{d'_i\tau}$.

   All this implies: $s\tau \mapsto_{R;r\tau} t\tau$.
2. $W \neq \emptyset$. Using 1. the proof is obvious.

4 Selection of Induction Schemes

To perform a proof by induction, it is necessary to provide some induction schemes. In our framework these schemes are defined *first* by a function which, given a conjecture, selects the positions of variables where induction will be applied and *second* by a special set of terms called a test set. In general the selection of good inductive positions leads to drastic improvements.

Let us consider first the problem of choosing the positions where variables need to be instanciated by induction schemes. In order to define the set of these variables, we introduce $VL(s)$, the set of linear variables of a term s and $Def(f)$, the set of terms with root symbol f.

Definition 4.1 *Given a rewrite system R on $T(F, X)$ the set of inductive positions for a function symbol f is: $Occ\_ind(R, f) = \{u/$ there exists $p \Rightarrow g \rightarrow d \in R$ such that $g \in Def(f)$, $u \in Occ(g) \setminus \{\epsilon\}$ and $g/u \notin VL(g)\}$.*

We say a variable x is *nullary* if there are only finitely many ground constructor terms with the same sort as x [2]. Given a term s, an *induction variable* of s is a variable that is nullary or that occurs at a position $u.v$ of s such that v is an inductive position of the root of s/u (if s is a variable then it is considered as an induction variable). Given a term s and a set of terms TS, σ is a *TS-substitution* for s, if it maps any induction variable of s to an element of TS of the same sort and any other variable to itself.

A rewrite rule $c \Rightarrow s \rightarrow r$ is *left-linear* if s is linear. A rewrite system R is *left-linear* if every rule in R is left-linear, otherwise R is said to be *non-left-linear*. A term is *strongly R-irreducible* if none of its non-variable subterms matches a left-hand side of R. If t is a term, then the *depth* of t is the maximum of the depths of the positions in t and denoted *depth(t)* or abusively $|t|$. The *strict depth* of t, written as *sdepth(t)*, is the maximum of the depths of the strict positions in t. The *depth of a rewrite system R*, denoted *depth(R)*, is defined as the maximum of the depths of the left-hand sides of R. Similarly, the *strict depth* of R, written a *sdepth(R)*, is the maximum of the depths of the strict positions in the left-hand sides of R. We define the number $D(R)$ to be $depth(R) - 1$ if $sdepth(R) < depth(R)$ and R is left linear, $depth(R)$ otherwise.

Instead of mapping induction variables to any terms we plan to instanciate them by some well-chosen elements that we call *test-sets*. A definition of test-sets for conditional theories was introduced in [Bouhoula *et al.*, 1992a]. It is possible to compute test sets for equational theories (see [Kounalis, 1990; Huber, 1991]). Unfortunately no algorithm exists for the general case of conditional theories.

Here we give a simpler definition of test-sets that will be sufficient for our purpose (we take advantage of the separation of symbols between defined and constructor operators).

Definition 4.2 *If R is a set of conditional rules then a test-set $S(R)$ for R is a finite set of R-irreducible constructor terms that has the following properties:*

a. *for any R-irreducible ground term s there exists a term t in $S(R)$ and a ground substitution σ such that $t\sigma = s$;*

b. *any non-ground term in $S(R)$ admits only non nullary variables and they may occur only at depth greater or equal than $D(R)$.*

The first property allows us to prove theorems by induction on the domain of irreducible terms rather than on the whole set of terms. Sets of terms with the property **a.** are usually called *cover sets* in the literature [Reddy, 1990; Zhang *et al.*, 1988]. However they cannot be used to refute theorems. The second property **b.** of test sets is fundamental for this purpose. It ensures that when an instance of a clause by an $S(R)$-substitution does not match any left-hand side of R this reveals an inconsistency (under some additional conditions to be detailed later).

The properties of test-sets that we shall need later on are listed in the next proposition.

Proposition 4.1 *We assume here that the constructors are free. For any term t and σ, $S(R)$-substitution for t we have:*

a. *$t\sigma$ does not contain an induction variable.*

b. *if $t\sigma$ is strongly R-irreducible then there exists τ such that $t\sigma\tau$ is ground and strongly R-irreducible.*

proof:
a. Without loss of generality, we can assume that t is a term $f(t_1, \ldots, t_n)$ where f is a defined operator and the t_i are elements of $T(C, X)$. Let l be *depth*(t) and let x be a variable in $t\sigma$ at occurrence u. We can assume that x does not occur in t, then by definition of test-sets we have $|u| > D(R)$. There are two cases to consider:

[2] this property can easily be decided when the constructors are free

- if R is left-linear and $sdepth(R) < depth(R)$. Then $|u| \geq depth(R)$. If u is an inductive position of f then there is a left-hand side of a rule where a function symbol occurs at position u. Since $sdepth(R) < depth(R)$ there is a rule whose left-hand side g satisfies $depth(g) > |u|$ and this contradicts $|u| \geq depth(R)$. Hence x cannot be an inductive variable of $t\sigma$.

- otherwise $|u| > depth(R)$. Hence u cannot be an occurrence of a left-hand side of a rule.

b. It is enough to prove the Claim: let t be a strongly R-irreducible term that does not contain any induction variable then there exists a ground instance $t\phi$ that is strongly R-irreducible. Assume that $Var(t) = \{x_1, \ldots, x_k\}$ and consider a ground substitution ϕ such that any $\phi(x)$ is strongly R-irreducible and:

$$\begin{array}{l} a. \forall\ i \in [1 \ldots k], |x_i\phi| > |t| \\ b. \forall\ i,j \in [1 \ldots k], i \neq j,\ \ ||x_i\phi| - |x_j\phi|| > |t| \end{array}$$

Since the variables x_i are not nullary it is possible to build such a substitution by choosing the $\phi(x_i)$ among the terms built from constructor symbols only. Assume now that $t\phi$ contains an instance of a left-hand side g of a rule in R. Now, since any $\phi(x_i)$ is strongly R-irreducible, there is a strict position u in t such that t/u is an instance of g. Let v be a position of g such that g/v is a function symbol. t/uv is a function symbol since t does not contain any induction variable. We consider two cases:

- Assume that g is linear. We can define a substitution σ such that for every variable x that occurs at position w of g we have $\sigma(x) = t/uw$. Such a substitution exists by linearity of g. We then have $t/u = g\sigma$ which is a contradiction with the assumption that t is strongly R-irreducible.

- Assume that g is non linear. Since t is not an instance of g (and $t/uw = g/w$ for every strict position w of g) there exist two occurrences u_1 and u_2 of a variable x in g such that: $t/uu_1 \neq t/uu_2$ and $t\phi/uu_1 = t\phi/uu_2$. There are three cases to be considered:

  a. if t/uu_1 and t/uu_2 are ground. In this case $t/uu_1 = t\phi/uu_1$ and $t/uu_2 = t\phi/uu_2$. Therefore $t/uu_1 = t/uu_2$, contradiction.

  b. if t/uu_1 is ground and t/uu_2 non ground. Then some x_i occurs in t/uu_2. We have $|x_i\phi| > |t|$ by construction of ϕ and therefore $|t\phi/uu_2| > |t|$. On the other hand $|t\phi/uu_2| = |t\phi/uu_1| = |t/uu_1| \leq |t|$. Contradiction.

  c. if t/uu_1 and t/uu_2 are non ground. Then there is an occurrence v and a variable x_k such that $t/uu_1v = x_k$ and $t/uu_2v \neq x_k$. If t/uu_2v is ground the proof is similar to $b.$. If t/uu_2v is non ground let $Var(t/uu_2v) = \{x_{i_1}, \ldots, x_{i_m}\}$. If $x_k \in Var(t/uu_2v)$ then $|t\phi/uu_1v| < |t\phi/uu_2v|$ and therefore we cannot have $t\phi/uu_1 = t\phi/uu_2$, contradiction. If $x_k \notin Var(t/uu_2v)$ then let x_j be the variable in $Var(t/uu_2v)$ such that $|x_j\phi| = max_{l=1,\ldots,m}|x_{j_l}\phi$.
  If $|x_k\phi| > |x_j\phi| + |t|$ then $|t\phi/uu_1v| = |x_k\phi| > |x_j\phi| + |t| > |t\phi/uu_2v|$ If $|x_j\phi| > |x_k\phi| + |t|$ then $|t\phi/uu_2v| \geq |x_j\phi| > |x_k\phi| + |t| > |t\phi/uu_1v| = |x_k\phi|$ and we derive a contradiction, too.

It is easy to compute test-sets in the case where all functions are completely defined:

Proposition 4.2 *If all defined operators are completely defined over free constructors then the set T of constructor terms (up to variable renaming) of depth $\leq D(R)$ where not nullary variables may occur only at depth $D(R)$ is a test-set for R.*

proof: since the defined operators are completely defined any ground irreducible terms is built only with constructors and therefore is an instance of an element of T. The second property of the definition is trivially verified.

The role of test sets for refutation is shown by the following definition that gives a falsity criteria for positive clauses:

Definition 4.3 *Suppose that we are given a rewrite system R and a test set $S(R)$. Then a clause $C \equiv g_1 = d_1 \vee \ldots \vee g_n = d_n$ is quasi-inconsistent with respect to R if there is a $S(R)$-substitution σ of C such that for all $1 \leq j \leq n$, $g_j\sigma \not\equiv d_j\sigma$ and the maximal elements of $\{g_j\sigma, d_j\sigma\}$ w.r.t. $\succ$ are strongly R-irreducible.*

The next theorem is analogous to one from [Kounalis and Rusinowitch, 1990a].

Theorem 4.1 *Let R be a ground convergent rewrite system with free constructors. If a positive clause C is quasi-inconsistent, then C is not an inductive consequence of R.*

proof: it is a direct consequence of proposition 4.1.

The following example was suggested by E. Kounalis:

Example 4.1 *Consider the "union" operator defined on lists of natural numbers built with constructors $\{cons,\ Nil\}$:*

$$\begin{array}{rcl} union(Nil, Nil) & \rightarrow & Nil \\ union(cons(x, Nil), Nil) & \rightarrow & cons(x, Nil) \\ union(Nil, cons(x, l)) & \rightarrow & cons(x, l) \\ union(cons(x, l), cons(y, l')) & \rightarrow & cons(x, cons(y, union(l, l'))) \end{array}$$

R is ground convergent and $S(R) = \{Nil,\ cons(x, l), 0, s(x)\}$. Consider the conjecture $union(l, l') = union(l', l)$. Among the test instances, we have: $union(cons(x, l), cons(y, l')) = union(cons(y, l'), cons(x, l))$ which simplifies to $cons(x, cons(y, union(l, l'))) = cons(y, cons(x, union(l, l')))$. Among the test instances of this last conjecture we have: $cons(x, cons(y, union(Nil, Nil))) = cons(y, cons(x, union(Nil, Nil)))$ which simplifies to $cons(x, cons(y, Nil)) = cons(y, cons(x, Nil))$, which is quasi-inconsistent (Both members are distinct, R-irreducible and do not contain any induction variable). So $union(l, l') = union(l', l)$ is not an inductive consequence of R. Note that with test sets, we can easily refute false conjectures with variables.

5 Automatic Case Analysis

We shall introduce now the *Case rewriting* relation that allows one to reduce goals with conditional rules without attempting to check their preconditions. The preconditions are appended to the goal as a context. Case rewriting can be viewed as an implementation of case analysis that is well adapted to the given conditional axioms. We have found this rule absolutely necessary for proving non trivial conjectures with our automatic system. Moreover it is the basis of a refutationally complete system for boolean systems. In this section we first discuss the problems with a former definition of case rewriting. Then we propose a new definition of case rewriting that is easier to automate and has given much better results on experiments. A first version of case rewriting was proposed in [Kounalis and Rusinowitch, 1990b]. Given a term l and a rule $p \Rightarrow g \rightarrow d$ such that g matches a subterm of l with substitution σ, this rewriting is expressed by the following inference rule:

$$l[g\sigma]_u \vee r \quad \vdash \quad l[d\sigma]_u \vee \neg p\sigma \vee r,\ \ l[g\sigma]_u \vee p\sigma \vee r$$

However an important control problem with this technique lies in the choice of the rule to apply during the proof by induction. This can be illustrated by the next example.

Example 5.1 *With R as in example 3.1. A test set here is* $\{0, s(0), s(s(x)), True, False\}$. *Let us prove:*

$$even(s(x)) = True \vee odd(x) = False \tag{18}$$

by the case rewriting rule above. Clause 18 can be split according to axiom 15 in the two following clauses:

$$even(x) = True \Rightarrow even(s(x)) = True \vee False = False \tag{19}$$
$$\neg(even(x) = True) \Rightarrow even(s(x)) = True \vee odd(x) = False \tag{20}$$

19 is a tautology and 20 is equivalent to:

$$even(x) = True \vee even(s(x)) = True \vee odd(x) = False \tag{21}$$

Now splitting clause 21 with axiom 16 yields:

$$even(x) = False \Rightarrow even(x) = True \vee even(s(x)) = True \vee True = False \tag{22}$$
$$\neg(even(x) = False) \Rightarrow even(x) = True \vee even(s(x)) = True \vee odd(x) = False \tag{23}$$

22 simplifies to:

$$even(x) = False \Rightarrow even(x) = True \vee even(s(x)) = True \tag{24}$$

that can be proved an inductive consequence of R. Clause 23 is equivalent to:

$$even(x) = False \vee even(x) = True \vee even(s(x)) = True \vee odd(x) = False \tag{25}$$

Note that the same case analysis can be applied infinitely often to 25. A possible way to avoid divergence is to limit application of the case analysis rule. Case rewriting in [Kounalis and Rusinowitch, 1990b] is controlled by conditions for avoiding infinite applications of the same rule. However, even when adding these technical conditions the proof of 25 diverges.

These problems motivated us to introduce a new case rewriting technique that rewrites a term t simultaneously to several terms $t_1, \ldots, t_n$ each reduction being respectively valid in different contexts $c_1, \ldots, c_n$. In other words, given a term t, we consider all the ways to rewrite it w.r.t. to axioms and positions. We must then prove that the disjunction DP of the conditions of the applied rules is inductively valid. Note that DP is usually a non Horn clause. Our approach to inductive proofs is non hierarchical: we can prove DP by simply adding it to the set of conjectures to be further processed. In the following definition CNF is a function that returns a conjunctive normal form of a universal formula. For instance, $CNF(P(x) \wedge Q(x)) = \{P(x), Q(x)\}$.

Definition 5.1 (Case rewriting) *Let R be a rewrite system and $C \equiv (a = b)^\epsilon \vee r$ be a clause. We define G as the set $\{< a[d\sigma]_u, P\sigma >$; there exists $P \Rightarrow g \to d$ in R, a position u in a such that $a/u = g\sigma$, $g\sigma$ is R-irreducible and does not contain an inductive variable$\}$. Then Case_rewriting $((a = b)^\epsilon, r)$ is the following set of clauses:*

$$\{\neg P \vee (a' = b)^\epsilon \vee r;\ < a', P > \in G\} \cup CNF(\bigvee_{<a',P>\in G} P)$$

Case rewriting can be applied with lemmas too. Note also that the conditions on $g\sigma$ in the definition of G are only required for the completeness of the procedure.

Example 5.2 (example 5.1 continued) *With our method, the proof of 18 is as follows: we apply case rewriting to get:*

$$even(x) = True \Rightarrow even(s(x)) = True \vee False = False \tag{26}$$
$$even(x) = False \Rightarrow even(s(x)) = True \vee True = False \tag{27}$$

We must prove:

$$even(x) = True \vee even(x) = False \tag{28}$$

26 is a tautology, 27 is simplified by R into:

$$even(x) = False \Rightarrow even(s(x)) = True \tag{29}$$

Instanciating x in 29 by elements 0 and $s(0)$ from the test set, yields clauses that are simplified by R and subsumed by an axiom. The instance of x in 29 by $s(s(y))$ gives a clause that is simplified by R and subsumed by 29, which becomes an induction hypothesis. In the same way 28 can easily be proved.

Example 5.3 *Consider the system R:*

$$\begin{array}{rcl} p(x,0,z) & \rightarrow & True \\ p(x,s(y),z) & \rightarrow & p(x,y,z) \\ p(y,x,z) = True \Rightarrow f(x,y,z) & \rightarrow & 0 \end{array}$$

To prove $f(x,y,z) = 0$ with the method of [Kounalis and Rusinowitch, 1990a], we instanciate x,y and z by 0 and $s(x')$ (from $S(R)$) in all possible ways. We obtain 8 equations and the proof of some of them diverges. With the method presented here, thanks to case rewriting, we do not need to consider all these inductive positions. We have: $Occ\_ind(p) = \{2\}$ and $Occ\_ind(f) = \emptyset$. To prove $f(x,y,z) = 0$, we apply case rewriting to get $p(y,x,z) = True \Rightarrow 0 = 0$ and $p(y,x,z) = True$. The first clause is a tautology and the second one is proved by instanciating x by 0 and $s(x)$.

Other authors have applied case rewriting techniques for inductive theorem proving. Among them [Bronsard and Reddy, 1990] and [Bevers, 1993] propose an approach related to ours but their methods cannot be considered as automatic, since they cannot check the applicability of case rewriting rules due to the fact that their provers are restricted to Horn clauses.

To conclude, our new case rewriting rule avoids many drawbacks of the previously defined ones and it allows to prove a larger class of theorems.

6 A Proof Procedure for Conditional Theories

6.1 Inferences rules

We present our procedure for proof by induction as a set of inference rules to be applied fairly to the goals. Let R be a rewrite system for the set of axioms Ax, we suppose that any defined function is completely defined. The procedure modifies incrementally two sets of clauses E and H, where E contains the conjectures to be checked and H contains clauses, previously in E, that have been reduced and can therefore be used as inductive hypotheses. This procedure is refutational in essence, and performs implicit induction w.r.t. to $\succ$. Its correctness is obtained by very simple arguments about the existence of a minimal counterexample. We think that our correctness proof is much simpler than the related ones [Reddy, 1990]. As a consequence it

generate: $(E \cup \{C\}, H) \vdash_I (E \cup (\cup_\sigma E_\sigma), H \cup \{C\})$

if $C \equiv (a = b)^\varepsilon \vee r$ and for every $S(R)$-substitution σ of C:
either $C\sigma$ is a tautology and $E_\sigma = \emptyset$
or $a\sigma \mapsto_{R[H \cup E];r\sigma} a'$ and $E_\sigma = \{(a' = b\sigma)^\varepsilon \vee r\sigma\}$
otherwise $E_\sigma = \mathit{case\_rewriting}((a = b)^\varepsilon\sigma, r\sigma)$

case simplify: $(E \cup \{(a = b)^\varepsilon \vee r\}, H) \vdash_I (E \cup E', H)$

if $E' = \mathit{case\_rewriting}((a = b)^\varepsilon, r)$

simplify: $(E \cup \{(a = b)^\varepsilon \vee r\}, H) \vdash_I (E \cup \{(a' = b)^\varepsilon \vee r\}, H)$

if $a \mapsto_{R[H \cup E];r} a'$ or $a[s]_u \mapsto_{H \cup E[R];r} a' \equiv a[t]_u$ and $u \neq \epsilon$

complement: $(E \cup \{\neg(a\sigma = b\sigma) \vee r\}, H) \vdash_I (E \cup \{(a\sigma = b'\sigma) \vee r\}, H)$

if $\neg(b = b') \in R$, $a = b \vee a = b' \in E \cup H$ and $b\sigma \succeq b'\sigma$.

delete: $(E \cup \{C\}, H) \vdash_I (E, H)$

if C is a tautology.

fail: $(E \cup \{C\}, H) \vdash_I \Box$

if for any (E', H'), $(E, H) \vdash_I (E', H')$ implies $C \in E'$

Figure 1: Inference System I

was easy for us to add many optimizations to the procedure and show that they do not affect correctness. The inference system for induction I contains the rules given in figure 1.

The *generate* rule allows to derive lemmas and initiates induction steps. The *case simplify* rule simplifies a conjecture with conditional rules and adds to the result the contexts where the respective reductions are valid. The *simplify* rule reduces a clause C with axioms from R, induction hypotheses from H and other conjectures (therefore we can simulate simultaneous induction). The premisses of C considered as a conditional axiom can also help to check that the preconditions of a rule being applied to C are valid. The *complement* rule transforms negative clauses to positive clauses that are easier to refute. The role of *deletion* is obvious. The *fail* rule is applied to (E, H) if no other rule can be applied to $C \in E$.

An I-derivation is a sequence of states:

$$(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots \vdash_I (E_n, H_n) \vdash_I \ldots$$

An I-derivation fails if it terminates with the rule *fail*.

In our former procedure of [Bouhoula *et al.*, 1992a] the *generate* rule was conditionned by ordering restrictions. The term a was bound to be a maximal member of an equation (w.r.t. $\succ$). The same restriction is valid in the actual setting and was omitted for clarity of presentation. Moreover the induction variables that have to be considered for building $S(R)$-substitutions can be selected out from maximal members of literals of the clauses. These additional constraints are crucial for avoiding divergence in many cases.

6.2 Correctness of the procedure

The correctness of I is obtained by defining a well-founded ordering on clauses and a notion of *fair derivation.* Fairness roughly means that every clause in the set of conjectures will be eventually modified by some inference.

Then we reason by contradiction: if a non valid clause is generated in a non failed derivation then a minimal one is generated too. We show that no inference step can apply to this clause. In other words, this clause persists in the derivation. This is a contradiction with the fairness hypothesis. The well-founded ordering on clauses is defined by first introducing the complexity of an equation. The complexity of an equation $g = h$ is defined as:

$$C(g=h) \;=\; \begin{cases} (\{g\},\{h\}) \; if \;\; g \succ h \\ (\{h\},\{g\}) \; if \;\; g \prec h \\ (\{g,h\},\bot) \; otherwise \end{cases}$$

where the new symbol $\bot$ is taken to be minimal in $\ll$. We define an ordering on equations as follows: $(a = b) \prec_e (c = d)$ iff $C(a = b)$ is smaller than $C(c = d)$ for the lexicographic composition of $\ll$ on the first and second components of the complexity. The multiset extension of $\prec_e$ will be denoted by $\prec\!\!\!\prec_e$.

Let C be a clause of type $\wedge_i a_i = b_i \Rightarrow \vee_j c_j = d_j$. We define $Rep(C) = \{C(a_i = b_i)\}_i \cup \{C(c_j = d_j)\}_j$. Given two clauses C_1, C_2, we say that $C_1 \prec_c C_2$ if lexicographically $Rep(C_1) \prec\!\!\!\prec_e Rep(C_2)$ or $nln(C_1) < nln(C_2)$, where $nln(C)$ is the number of negative literals of C.

Example 6.1 *Let $C_1 \equiv \neg(c = d) \vee a = b$ and $C_2 \equiv a' = b \vee c' = d'$ with $a \divideontimes b$, $a \divideontimes a'$, $a' \prec b$, $c \succ d, c' \succ d'$ and $c' \prec c$ then $C_2 \prec_c C_1$.*

The correctness of a procedure based on our inference system relies on a fairness assumption: every conjecture to be checked must be considered at some step. More formally, a derivation $(E_0, H_0) \vdash_I (E_1, H_1) \vdash_I \ldots$ is *fair* if either it fails or it is infinite and the set of persisting clauses $(\cup_{i \geq 0} \cap_{j \geq i} \; E_j)$ is empty.

Lemma 2 *Let $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots$ be a fair I-derivation such that $R \not\models_{ind} E_0$ then there exists i such that the rule $fail$ applies to (E_i, H_i).*

proof: Let $C\theta$ be a minimal element w.r.t. $\prec_c$ of the set $\{D\sigma / D \in \cup_i E_i$ and there is a ground R-irreducible substitution σ such that $R \not\models_{ind} D\sigma\}$. C exists since $R \not\models_{ind} E_0$ and $\prec_c$ is well-founded. Assume that $C \in E_j$. Then there exists $k \geq j$ such that the rule *fail* applies to (E_k, H_k). It is sufficient to check that C cannot be simplified nor deleted, and that neither *generate* nor *complement* apply to C. As a consequence *fail* applies since the clause C must not persist in the derivation by the fairness hypothesis. The detailed proof is found in Appendix 2.

The next theorem is a straightforward consequence of the above lemma:

Theorem 6.1 (correctness) *Let R be a rewrite system and let $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots$ be a fair I-derivation. If it does not fail then $R \models_{ind} E_0$.*

Since every I-derivation from $(E, \emptyset)$ to $(\emptyset, H)$, where H is some set of clauses, is fair then the conjectures of E are inductive consequences of R. This remark is important from a practical point of view. Note also that E_0 is valid even when the derivation is infinite.

6.3 Refutation of conjectures

From now on, we assume that R is a ground convergent rewrite system such that all its defined symbols are completely defined. Moreover, if the defined function g appears in a l.h.s. of a conditional rule, then every rule that contains g in its l.h.s. is linear. Finally, we assume that every left-hand side of a conditional rule has a defined symbol. Let us call J the set of inference rules obtained by adding to I the rule:

disproof: $(E \cup \{C\}, H) \vdash_J$ *Disproof* if C is quasi-inconsistent.

A J-derivation fails if it ends with *fail.* If *disproof* is applied then a quasi-inconsistent clause is detected and therefore, from theorem 4.1, we can conclude that some conjecture is false:

Corollary 1 *Let $(E_0, \emptyset) \vdash_J (E_1, H_1) \vdash_J \ldots$ be a J-derivation. If there is a k such that disproof applies to (E_k, H_k) then $R \not\models_{ind} E_k$.*

If at step k, we find that $R \not\models_{ind} E_k$, we can conclude that E_0 is not valid either in R. This is a consequence of the next result:

Lemma 3 *Let $(E_0, \emptyset) \vdash_J (E_1, H_1) \vdash_J \ldots$ be a J-derivation. If for all j such that $j < i$ we have $R \models_{ind} E_j$ then $R \models_{ind} E_i$.*

proof: If $(E_{k-1}, H_{k-1}) \vdash_J (E_k, H_k)$ by *generate* on C, consider $C\sigma$ which is an instance of C by a $S(R)$-substitution. If $C\sigma$ is simplified then every auxiliary equation which is used for rewriting is either in R or $E_{j'}$ $(j' < k)$ and hence E_k is valid in R. If *case_rewriting* is applied to $C\sigma$ then E_k is valid in R since all defined symbols are completely defined, by hypothesis. For *case simplify* the argument is the same as above.

If $(E_{k-1}, H_{k-1}) \vdash_J (E_k, H_k)$ by *simplify* then the equations which are used for simplification occur in some E_j $(j < k)$ and therefore are valid in R by hypothesis. Hence, E_k is valid too in R.

If $(E_{k-1}, H_{k-1}) \vdash_J (E_k, H_k)$ by *complement*. Suppose that *complement* is applied to $C \equiv \neg(a\theta = b\theta) \vee r$ to obtain $C' \equiv (a\theta = b'\theta) \vee r$. We can show that $R \models_{ind} C'$. The detailed proof is found in Appendix 2.

Theorem 6.2 *Let R be a ground convergent rewrite system such that every defined symbol is completely defined and let $(E_0, \emptyset) \vdash_J (E_1, H_1) \vdash_J \ldots$ be a J-derivation. If there exists j such that* disproof *applies to (E_j, H_j) then $R \not\models E_0$.*

proof: Let $(E_0, \emptyset) \vdash_J (E_1, H_1) \vdash_J \ldots$ be a J-derivation. Assume that there is j such that *disproof* applies to (E_j, H_j). It is clear from corollary 1 that $R \not\models_{ind} E_j$ and by lemma 3, we have $R \not\models_{ind} E_0$.

7 A refutationally complete procedure for theories with boolean preconditions

In this section we shall consider axioms that are conditional rules with boolean preconditions over free constructors. To be more specific, we assume there exists a sort *bool* with two nullary free constructors $\{True,\ False\}$. Every rule in R is of type: $\wedge_{i=1}^{n} p_i = p'_i \Rightarrow s \rightarrow t$ where for all i in $[1 \ldots n]$, $p'_i \in \{True,\ False\}$. Such a system R is called a *boolean rewrite system.* For $\alpha \in \{True,\ False\}$ we denote by α^- the complementary *bool* symbol of α. Conjectures will be *boolean clauses*, that is clauses whose negative literals are of type $\neg(p = p')$ where $p' \in \{True, False\}$.

We also assume that any function symbol p with boolean values is completely defined. In other words, the following is inductively valid:

$$p(\vec{x}) = True \vee p(\vec{x}) = False$$

Therefore, the following propositions are also valid and they can be used to eliminate negations:

$$\neg(p(\vec{x}) = True) \Leftrightarrow (p(\vec{x}) = False), \quad \neg(p(\vec{x}) = False) \Leftrightarrow (p(\vec{x}) = True)$$

We can then define a new inference system K from I by reformulating *complement* as follows:

complement: $(E \cup \{\neg(a = \alpha) \vee r\}, H) \vdash_K (E \cup \{(a = \alpha^-) \vee r\}, H)$ if $\alpha \in \{True,\ False\}$.

and replacing the *fail* rule by:

disproof: $(E \cup \{C\}, H) \vdash_K$ *Disproof* if for any (E', H'), $(E, H) \vdash_K (E', H')$ implies $C \in E'$.

A K-derivation *fails* if it ends with *disproof.* The inference system K allows to refute false conjectures, thanks to the following results:

Lemma 4 *Let $C \equiv (a = b)^\epsilon \vee r$. If a contains a defined symbol then generate can be applied to $(E \cup \{C\}, H)$.*

proof: Let $C \equiv (a = b)^\epsilon \vee r$ and let σ be an $S(R)$-substitution of C. The term a contains a term s of the form $f(t_1, \ldots, t_n)$ where f is a defined symbol and for all i, t_i is in $T(C, X)$. The term $s\sigma$ matches a left-hand side of R, otherwise there exists a substitution τ such that $s\sigma\tau$ is ground and strongly irreducible by R, by using clause **b.** of proposition 4.1 . This leads to a contradiction since we have assumed that f is completely defined. On the other hand, σ is an $S(R)$-substitution of C, therefore $s\sigma$ does not contain an inductive variable. So either some inductive rewriting or some case rewriting can be applied to $C\sigma$ and therefore *generate* can be applied to C. This ends the proof of lemma 4.

If *disproof* is applied in a K-derivation, then there exists a boolean clause C such that *generate* cannot be applied to C. Therefore there exists a $S(R)$-substitution σ such that $C\sigma$ is not a tautology. Moreover by lemma 4, $C\sigma$ cannot contain a defined symbol. Hence it only contains constructor symbols and therefore, since constructors are free by hypothesis, it is strongly R-irreducible. As a consequence, C is a quasi-inconsistent clause.

Note now that the only rule that permits to introduce negative clauses is *case_rewriting.* Let us assume that E_0 only contains boolean clauses. Since the axioms have boolean preconditions, all the clauses generated in a K-derivation are boolean. So the new inference system K can be proved refutationally complete for boolean clauses as well.

Theorem 7.1 *Let R be a ground convergent boolean rewrite system such that every defined symbol is completely defined and let $(E_0, \emptyset) \vdash_K (E_1, H_1) \vdash_K \ldots$ be a fair K-derivation such that E_0 only contains boolean clauses. Then $R \not\models_{ind} E_0$ iff the derivation fails.*

8 Optimizations

In this section, we shall enhance our inference system J by new simplification rules to handle non orientable equations and gain efficiency. In particular, we allow under some conditions a term s to be rewritten to another one that is not comparable with s. For this purpose we define a new rewrite relation that we call *relaxed inductive rewriting* :

Definition 8.1 (relaxed inductive rewriting) *Let R and W be two sets of conditional equations. Consider a clause $C \equiv (a = b)^{\epsilon} \vee r$ and its skolemized version $\overline{C} \equiv (\overline{a} = \overline{b})^{\epsilon} \vee \overline{r}$. We write:*
$a \rightsquigarrow_{R[W];r} a'$ if:
either *$\overline{a} \rightarrow_{prem(\overline{r})} \overline{a'}$ and $a \succ a'$,*
or *there exists a position u in a, a substitution σ and a conditional equation $\wedge_{i=1}^{n} a_i = b_i \;\Rightarrow s = t$ in R such that:*

1. *$a \equiv a[s\sigma]_u$ and $a' \equiv a[t\sigma]_u$.*

2. *$\{a[s\sigma]_u\} \gg \{a_1\sigma,\; b_1\sigma,\; \ldots\;, a_n\sigma,\; b_n\sigma\}$.*

3. *$\forall i \in [1..n]\;\; \exists c'_i, d'_i$ such that $a_i\sigma \mapsto^{*}_{R\cup W\;[\;];r} c'_i$ and $b_i\sigma \mapsto^{*}_{R\cup W\;[\;];r} d'_i$ and $\overline{c'_i} =_{prem(\overline{r})} \overline{d'_i}$.*

Note that lemma 1 remains valid if we replace $\mapsto_{R[W];r}$ by $\rightsquigarrow_{R[W];r}$.

Consider now the inference N obtained by adding to J the following rules:

right simplify of constructors: $(E \cup \{f(\vec{s}) = f(\vec{t}) \vee r\}, H) \quad \vdash_N \quad (E \cup (\cup_i\{s_i = t_i \vee r\}_i), H)$
if f is a free constructor.

left simplify of constructors: $(E \cup \{\neg f(\vec{s}) = f(\vec{t}) \vee r\}, H) \quad \vdash_N \quad (E \cup \{\vee_i \neg(s_i = t_i) \vee r\}, H)$
if f is a free constructor.

subsumption: $(E \cup \{C\}, H) \vdash_N \; (E, H)$
if C is subsumed by another clause of $R \cup H \cup E$.

and replacing the simplify rule by:

simplify: $(E \cup \{(a = b)^{\epsilon} \vee r\}, H) \vdash_N \; (E \cup \{(a' = b)^{\epsilon} \vee r\}, H)$
if $a \rightsquigarrow_{R[H\cup E];r} a'$ or $a[s]_u \mapsto_{H\cup E[R];r} a' \equiv a[t]_u$, $u \neq \epsilon$ and $a' = b \prec_e a = b$.

Simplify makes possible to simplify non orientable equations derived in H or E, such as commutativity, when standard inductive rewriting fails. *Left simplification of constructors* and *right simplification of constructors* take advantage that constructors are free to decompose terms. *Subsumption* delete clauses C subsumed by an element of R or $H \cup (E \setminus \{C\})$.

Note that lemma 2 remains true when we replace I-derivations by N-derivations. Therefore theorem 6.1 is valid and the inference system N is correct. Refutational completeness is also preserved for boolean systems. The detailed proof is found in Appendix 2.

9 Computer Experiments

Our prototype SPIKE (written in Caml Light) is designed to prove the validity of a set of clauses in a conditional theory. The first step in a proof session is to check if all defined functions are completely defined. If this step is successful then we can use a more efficient version of the case rewriting rule. The second step is to check the ground convergence of the set of axioms. If the first two steps are successful then we can refute false conjectures. The third step is to compute test sets and inductive positions. After these preliminary tasks, the proof starts.

Example 9.1 *Consider example 2.1 and let us prove the conjecture* $sorted(insert(x1, x2)) = sorted(x2)$. *Note that RRL [Zhang* et al., *1988] is unable to succeed with this example unless the user suggests some well-chosen lemmas.*

```
All functions are completely defined;

R is ground convergent;

test set of R:

 -> nat = {0, s(0),s(s(x1))}
 -> list = {Nil, cons(0,Nil), cons(s(x1),Nil), cons(0,cons(x1,x2)), cons(s(x1),cons(x2,x3))}
 -> bool = {True, False}

induction positions of functions:

 -> sorted: [[1];[1;2]]
 -> insert: [[2]]
 -> <=: [[1];[2]]

E0 = {sorted(insert(x1,x2)) = sorted(x2)}


H0 = {}


Application of generate on:
    sorted(insert(x1,x2)) = sorted(x2):
 1) True = sorted(Nil) ;
 2) x1<=0 = True => sorted(Cons(x1,Cons(0,Nil))) = sorted(Cons(0,Nil)) ;
 3) x1<=0 = False => sorted(Cons(0,insert(x1,Nil))) = sorted(Cons(0,Nil)) ;
 4) x1<=S(x2) = True => sorted(Cons(x1,Cons(S(x2),Nil))) = sorted(Cons(S(x2),Nil)) ;
 5) x1<=S(x2) = False => sorted(Cons(S(x2),insert(x1,Nil))) = sorted(Cons(S(x2),Nil)) ;
 6) x1<=0 = True => sorted(Cons(x1,Cons(0,Cons(x2,x3)))) = sorted(Cons(0,Cons(x2,x3))) ;
 7) x1<=0 = False => sorted(Cons(0,insert(x1,Cons(x2,x3)))) = sorted(Cons(0,Cons(x2,x3))) ;
 8) x1<=S(x2) = True =>
        sorted(Cons(x1,Cons(S(x2),Cons(x3,x4)))) = sorted(Cons(S(x2),Cons(x3,x4))) ;
 9) x1<=S(x2) = False =>
        sorted(Cons(S(x2),insert(x1,Cons(x3,x4)))) = sorted(Cons(S(x2),Cons(x3,x4)))

...


Delete
x1<=S(x2) = True => sorted(Cons(x1,Cons(S(x2),Cons(x3,x4)))) = sorted(Cons(S(x2),Cons(x3,x4)))
 it is subsumed by:
x1<=x2 = True => sorted(Cons(x1,Cons(x2,x3))) = sorted(Cons(x2,x3)) of R

...


E17 = {x1<=S(x2) = True, S(x2)<=x1 = True ;
      x1<=0 = True, sorted(Cons(x2,insert(x1,x3))) = sorted(Cons(x2,x3)), x1<=x2 = True ;
```

```
       x1<=S(x2) = True, sorted(Cons(S(x2),
          Cons(x3,insert(x1,x4)))) = sorted(Cons(S(x2),Cons(x3,x4))), x1<=x3 = True ;
       x1<=S(x2) = True, x1<=x3 = False,
          sorted(Cons(x3,x4)) = sorted(Cons(S(x2),Cons(x3,x4))), S(x2)<=x1 = False}

H17 = {sorted(insert(x1,x2)) = sorted(x2)}

Application of case rewriting using R on:
    x1<=S(x2) = True, x1<=x3 = True,
       sorted(Cons(S(x2),Cons(x3,insert(x1,x4)))) = sorted(Cons(S(x2),Cons(x3,x4))):
 1) S(x2)<=x3 = False => x1<=S(x2) = True,
    x1<=x3 = True, sorted(Cons(S(x2),Cons(x3,insert(x1,x4)))) = False ;
 2) S(x2)<=x3 = True => x1<=S(x2) = True, x1<=x3 = True,
    sorted(Cons(S(x2),Cons(x3,insert(x1,x4)))) = sorted(Cons(x3,x4))

...

Simplification of:

    S(x2)<=x3 = True  => x1<=S(x2) = True, x1<=x3 = True,
         sorted(Cons(S(x2),Cons(x3,insert(x1,x4)))) = sorted(Cons(x3,x4))
 by R[H20 U E20];r:
    x1<=S(x2) = True, S(x2)<=x3 = False,
       x1<=x3 = True, sorted(Cons(x3,insert(x1,x4))) = sorted(Cons(x3,x4))

...

E249 = {}

The initial conjectures are inductive consequences of R
```

The following sub lemmas have been generated automatically during the proof and have played a role in it:

```
{x1<=x2 = True, S(x1)<=x3 = False, x2<=x1 = False, S(x2)<=x3 = True ;
 x1<=x2 = False, x3<=S(x1) = False, x3<=S(x2) = True, S(x1)<=x3 = True ;
 sorted(Cons(x1,insert(S(S(x2)),x3))) = sorted(Cons(x1,x3)), S(S(x2))<=x1 = True ;
 sorted(Cons(x1,insert(S(0),x2))) = sorted(Cons(x1,x2)), S(0)<=x1 = True ;
 x1<=x2 = True, x2<=x1 = True ;
 x1<=S(x2) = True, x1<=x3 = False,
         S(x2)<=x1 = False, sorted(Cons(x3,x4)) = False, S(x2)<=x3 = True ;
 x1<=S(x2) = True, x1<=x3 = True,
         sorted(Cons(x3,insert(x1,x4))) = sorted(Cons(x3,x4)), S(x2)<=x3 = False ;
 x1<=0 = True, sorted(Cons(x2,insert(x1,x3))) = sorted(Cons(x2,x3)), x1<=x2 = True ;
 x1<=S(x2) = True, S(x2)<=x1 = True ;
 sorted(insert(x1,x2)) = sorted(x2)}
```

10 Conclusion

We have proposed a new procedure for proof by induction in conditional theories. Our procedure relies on the implicit induction paradigm and puts the stress on simplification and case analysis. As our previous procedure [Bouhoula *et al.*, 1992a], it allows simplification of conjectures by conjectures and has been extended to handle non-orientable equations. It can also refute non valid conjectures. A main contribution of this paper is that our strategy is refutationally complete for a class of rewrite systems that can specify numerous interesting examples. This class contains the boolean ground convergent rewrite systems with completely defined functions over free constructors. In other words with our procedure every false conjecture will be disproved in finite time. However, our method remains valid even when the functions are not completely defined. Note that our correctness and completeness proofs do not require an elaborated notion of fairness.

We plan to enhance the system with generalisation techniques for suggesting lemmas, as the one proposed in [Basin and Walsh, 1993], since for many examples the "Generate" rule is not sufficient for deriving the lemmas needed for achieving a goal. Also extension of the method to parametrized specifications should lead to shorter and structured proofs. The format of implicit induction is not user-friendly. Therefore, some effort should be devoted to understand the relationship between our method and explicit induction (in particular, in the case of mutual induction [Walther, 1993]). We shall explore the possibility of translations between the two frameworks. If such a translation is available then we should benefit from recent promising works on proof planning [Ireland, 1992].

An extension to theories that are presented by stratified sets of clauses should also follow easily from our work.

Acknowledgements: We thank Emmanuel Kounalis, Hélène Kirchner and Maria Huber for their valuable comments.

References

[Bachmair, 1988] L. Bachmair. *Proof by consistency in equational theories.* In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 228–233, 1988.

[Basin and Walsh, 1993] D. Basin and T. Walsh *Difference Unification.* In *Proceedings of 13th IJCAI*, Morgan Kaufmann Publishers, volume 1, pages 116–122, 1993

[Bevers, 1993] E. Bevers. *Automated Reasoning in Conditional Algebric Specifications: Termination and Proof by Consistency.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 1993.

[Bouhoula *et al.*, 1992a] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. Technical Report 1636, INRIA, 1992.

[Bouhoula *et al.*, 1992b] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Spike: an automatic theorem prover. In *Proceedings of LPAR'92*, volume 624 of *LNAI*, Saint Petersbourg, Russia, July 1992. Springer-Verlag.

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

[Bronsard and Reddy, 1990] F. Bronsard and S. Reddy. Conditional rewriting in focus. In S. Kaplan and M. Okada, editors, *Proceedings of the 2nd Workshop on Conditional and Typed Rewriting Systems*, volume 516 of *LNCS*. Springer-Verlag, 1990.

[Bundy *et al.*, 1989] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 132–146. Springer-Verlag, July 1989.

[Chadha and Plaisted, 1992] R. Chadha and D.A. Plaisted. Mechanizing mathematical induction. *Presented at the Second International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida*, 1992.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leuven, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers North-Holland, 1990.

[Dershowitz *et al.*, 1988] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings 9th International Conference on Automated Deduction, Argonne (Ill., USA)*, volume 310 of *LNCS*. Springer-Verlag, May 1988.

[Fribourg, 1986] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proceedings 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *LNCS*, pages 105–115. Springer-Verlag, 1986.

[Guttag, 1978] J. Guttag. Abstract data types and software validation. *Communications of the ACM*, volume 21, pages 1048–1064, 1978.

[Huber, 1991] M. Huber. Test-set approaches for ground reducibility in term rewriting systems, characterizations and new applications. Master's thesis, Technische Universität Berlin, 1991.

[Huet and Hullot, 1982] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, October 1982.

[Huet, 1991] G. Huet. The Gilbreath trick: a case study in axiomatisation and proof development in the coq proof assistant. Technical Report 1511, INRIA, 1991.

[Ireland, 1992] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, volume 624 of *LNAI*. Springer-Verlag, 1992.

[Jouannaud and Kounalis, 1986] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 358–366, 1986.

[Kapur and Musser, 1987] D. Kapur and D. Musser. Proof by consistency Artificial Intelligence, volume 31(2), pages 125–157, 1987

[Kounalis and Rusinowitch, 1990a] E. Kounalis and M. Rusinowitch. A mechanization of conditional reasoning. In *First International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida*, January 1990.

[Kounalis and Rusinowitch, 1990b] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proceedings of the American Association for Artificial Intelligence Conference, Boston*, pages 240–245. AAAI Press and MIT Press, July 1990.

[Kounalis, 1990] E. Kounalis. Testing for inductive (co)-reducibility. In A. Arnold, editor, *Proceedings 15th CAAP, Copenhagen (Denmark)*, volume 431 of *LNCS*, pages 221–238. Springer-Verlag, May 1990.

[Musser, 1980] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings 7th ACM Symp. on Principles of Programming Languages*, pages 154–162. Association for Computing Machinery, 1980.

[Padawitz, 1988] P. Padawitz. *Computing in Horn Clause Theories*, Springer Verlag, 1988.

[Reddy, 1990] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *LNCS*, pages 162–177. Springer-Verlag, 1990.

[Walther, 1993] C. Walther. *Combining Induction Axioms By Machine.* In *Proceedings of 13th IJCAI*, Morgan Kaufmann Publishers, volume 1, pages 95–100, 1993

[Zhang *et al.*, 1988] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction, Argonne (Ill., USA)*, volume 310 of *LNCS*, pages 162–181. Springer-Verlag, 1988.

[Zhang, 1993] H. Zhang Implementing Contextual Rewriting *Proceedings of the 3rd CTRS Workshop*, volume 656 of *LNCS*. Springer-Verlag, 1993.

Appendix 1: The Gilbreath Card Trick

A Introduction

Suppose you have a deck of cards of even length. Suppose the cards alternate between red ones and black ones. Cut the deck into two piles, a and b. Shuffle a and b together. Then the following is true of the shuffled deck. If the bottom-most cards in a and b are of different color, then when the cards of the shuffled deck are taken from the top in adjacent pairs, each pair contains a card of each color. On the other hand, if the bottom-most cards in a and b are the same color, the above pairing property holds after rotating the shuffled deck by one card, i.e., moving the bottom card to the top.

An interactive proof of Gilbreath Cards Trick was first given by G. Huet [Huet, 1991] using the COQ proof assistant. B. Boyer has used NQTHM to derive another proof. A similar but much faster proof was obtained by H. Zhang with RRL. These two automatized proofs require many lemmas, some of them being non-obvious. For instance, B. Boyer introduces a predicate "silly" that is "only defined to force a certain weird induction" (here we quote B. Boyer). The same predicate appears in Zhang's experiment. On the other hand, our proof is based on 5 lemmas easy to understand. These lemmas have been suggested by a first unsuccessful proof attempt with SPIKE. The source of failure was identified by the impossibility of reducing a family of patterns. Hence we have introduced the adequate lemmas for simplifying them. This was enough to derive a proof. Note that if we use the same formulation as H. Zhang, our system needs only 2 lemmas to prove the Gilbreath Cards Trick (see section D).

The proof has taken more CPU time than Boyer's (and Zhang's). However, it is difficult to compare these approaches from the efficiency point of view, since our we have spent very few time to get the right lemmas. On the other hand, the differences between programming languages lead also to some discrepancy in the performance.

The following array compare users inputs for the proof of Gilbreath Card Trick with NQTHM, RRL and SPIKE:

	NQTHM	RRL		SPIKE	
Definitions	?	10	10	9	10
Induction scheme definition	1	1	0	0	0
Lemmas	17	23	8	5	2

In chronological order we have: NQTHM, first column of RRL, first column of SPIKE, second column of RRL, corresponding, to more recent experiments obtained after modification of the code of RRL, second column of SPIKE corresponding to same inputs as RRL.

B Formulation of the problem in SPIKE

In this section, we define in SPIKE various predicates and functions to formulate the Gilbreath Card Trick:

There are two kind of cards: R=red and B=black.

$R :\rightarrow card$

$B :\rightarrow card$

The constructors for lists of cards:

$Null :\rightarrow list$

$Cons : card \times list \rightarrow list$

The constructors for boolean:

$True :\rightarrow bool$

$False :\rightarrow bool$

We suppose that $True <> False$:

$True = False \Rightarrow$

$[neg : card \rightarrow card]$ The negation function maps R to B and conversely:

$neg(R) = B$

$neg(B) = R$

$[paired : card \times card \rightarrow bool]$ $paired(x, y) = true$ iff x and y are cards of opposite color.

$paired(R, B) = True$

$paired(B, R) = True$

$paired(x, x) = False$

$[append : list \times list \rightarrow list]$ appends two lists.

$append(Null, y) = y$

$append(Cons(x, y), z) = Cons(x, append(y, z))$

$[rotate : list \rightarrow list]$ rotate the first element to the end of a list.

$rotate(Null) = Null$

$rotate(Cons(x, y)) = append(y, Cons(x, Null))$

$[even : list \rightarrow bool]$ $even(x) = true$ iff x is of even length.

$even(Null) = True$

$even(Cons(x, Null)) = False$

$even(Cons(x1, Cons(x2, y))) = even(y)$

$[opposite : list \times list \rightarrow bool]$ $opposite(x, y) = true$ iff the first cards of x and y, respectively, are of opposite color.

$opposite(Null, y) = False$

$opposite(x, Null) = False$

$opposite(Cons(x1, y1), Cons(x2, y2)) = paired(x1, x2)$

$[pairedlist : list \rightarrow bool]$ $pairedlist(x) = true$ iff x is a list of cards such that if we repeatedly take two cards from its top, the two cards are found to be of opposite color.

$pairedlist(Null) = True$

$pairedlist(Cons(x, Null)) = True$

$paired(x1, x2) = True \Rightarrow pairedlist(Cons(x1, Cons(x2, x))) = pairedlist(x)$

$paired(x1, x2) = False \Rightarrow pairedlist(Cons(x1, Cons(x2, x))) = False$

$[alter : list \rightarrow bool]$ $alter(x) = true$ iff x is a list of cards whose colors alternate.

$alter(Null) = True$

$alter(Cons(x1, Null)) = True$

$paired(x1, x2) = True \Rightarrow alter(Cons(x1, Cons(x2, x))) = alter(Cons(x2, x))$

$paired(x1, x2) = False \Rightarrow alter(Cons(x1, Cons(x2, x))) = False$

$[shuffle : list \times list \times list \rightarrow bool]$ $shuffle(x, y, z) = true$ iff z is a merge of x and y.

$shuffle(Null, Null, Null) = True$

%% $shuffle(x, Null, z) \Leftrightarrow (x = z)$

$shuffle(Null, Null, Cons(x3, y3)) = False$

$shuffle(Cons(x1, y1), y2, Null) = False$

$paired(x1, x3) = True \Rightarrow shuffle(Cons(x1, y1), Null, Cons(x3, y3)) = False$

$paired(x1, x3) = False \Rightarrow shuffle(Cons(x1, y1), Null, Cons(x3, y3)) = shuffle(y1, Null, y3)$

%% $shuffle(Null, x, z) \Leftrightarrow (x = z)$

$shuffle(y1, Cons(x2, y2), Null) = False$

$$\mathit{paired}(x2,x3) = \mathit{True} \Rightarrow \mathit{shuffle}(\mathit{Null},\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{False}$$

$$\mathit{paired}(x2,x3) = \mathit{False} \Rightarrow \mathit{shuffle}(\mathit{Null},\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{shuffle}(\mathit{Null},y2,y3)$$

$$\%\% \ \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) \Leftrightarrow$$
$$\%\% \ ((x1 = x3) \ \mathit{and} \ \mathit{shuffle}(y1,\mathit{Cons}(x2,y2),y3)) \ \mathit{or} \ ((x2 = x3) \ \mathit{and} \ \mathit{shuffle}(\mathit{Cons}(x1,y1),y2,y3))$$

$$\mathit{paired}(x1,x3) = \mathit{False}, \mathit{shuffle}(y1,\mathit{Cons}(x2,y2),y3) = \mathit{True} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{True}$$

$$\mathit{paired}(x2,x3) = \mathit{False}, \mathit{shuffle}(\mathit{Cons}(x1,y1),y2,y3) = \mathit{True} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{True}$$

$$\mathit{paired}(x1,x3) = \mathit{True}, \mathit{paired}(x2,x3) = \mathit{True} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{False}$$

$$\mathit{shuffle}(y1,\mathit{Cons}(x2,y2),y3) = \mathit{False}, \mathit{shuffle}(\mathit{Cons}(x1,y1),y2,y3) = \mathit{False} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{False}$$

$$\mathit{paired}(x1,x3) = \mathit{True}, \mathit{shuffle}(\mathit{Cons}(x1,y1),y2,y3) = \mathit{False} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{False}$$

$$\mathit{paired}(x2,x3) = \mathit{True}, \mathit{shuffle}(y1,\mathit{Cons}(x2,y2),y3) = \mathit{False} \Rightarrow \mathit{shuffle}(\mathit{Cons}(x1,y1),\mathit{Cons}(x2,y2),\mathit{Cons}(x3,y3)) = \mathit{False}$$

The main theorem

Imagine x and y to be the two card stacks that result from cutting the original deck. The original deck is thus "append(x,y)". Suppose that the original deck is of alternating color and of even length. Suppose further that z is a shuffle of x and y. If x and y start with cards of opposite color, then z satisfies "pairedlist".

$$\mathit{alter}(\mathit{append}(u,v)) = \mathit{True}, \mathit{even}(\mathit{append}(u,v)) = \mathit{True}, \mathit{opposite}(u,v) = \mathit{True}, \mathit{shuffle}(u,v,w) = \mathit{True}$$
$$\Rightarrow \mathit{pairedlist}(w) = \mathit{True}$$

On the other hand, if x and y start with cards of same color, then the result of moving the top card of z to the end of z satisfies "pairedlist":

$$\mathit{alter}(\mathit{append}(u,v)) = \mathit{True}, \mathit{even}(\mathit{append}(u,v)) = \mathit{True}, \mathit{opposite}(u,v) = \mathit{False}, \mathit{shuffle}(u,v,w) = \mathit{True}$$
$$\Rightarrow \mathit{pairedlist}(\mathit{rotate}(w)) = \mathit{True}$$

C Properties of the specification

C.1 Checking the completeness of the specification

SPIKE checks automatically if an operator f in a specification is completely defined. The program builds a pattern tree for f. The leaves of the tree give a partition of the possible arguments for f. Then if all the leaves are reducible, the answer is affirmative.

The test is successful on our axioms. This is shown during the first part of the proof session which appears below.

```
          --------------------------------------
          >>> Display of pattern tree of neg  <<<
          --------------------------------------

 neg(x1)
  neg(R)  ok
  neg(B)  ok

          ------------------------------------------
          >>> Display of pattern tree of paired  <<<
          ------------------------------------------

paired(x1,x2)
  paired(R,x2)
    paired(R,R)  ok
    paired(R,B)  ok
  paired(B,x2)
```

```
    paired(B,R)  ok
    paired(B,B)  ok

            ------------------------------------------
            >>> Display of pattern tree of append  <<<
            ------------------------------------------

append(x1,x2)
  append(Null,x2)  ok
  append(Cons(x3,x4),x2)  ok
            ------------------------------------------
            >>> Display of pattern tree of rotate  <<<
            ------------------------------------------

rotate(x1)
  rotate(Null)  ok
  rotate(Cons(x2,x3))  ok
            ----------------------------------------
            >>> Display of pattern tree of even  <<<
            ----------------------------------------

even(x1)
  even(Null)  ok
  even(Cons(x2,x3))
    even(Cons(x2,Null))  ok
    even(Cons(x2,Cons(x1,x4)))  ok


            --------------------------------------------
            >>> Display of pattern tree of opposite  <<<
            --------------------------------------------

opposite(x1,x2)
  opposite(Null,x2)  ok
  opposite(Cons(x3,x4),x2)
    opposite(Cons(x3,x4),Null)  ok
    opposite(Cons(x3,x4),Cons(x1,x5))  ok
            ----------------------------------------------
            >>> Display of pattern tree of pairedlist  <<<
            ----------------------------------------------

pairedlist(x1)
  pairedlist(Null)  ok
  pairedlist(Cons(x2,x3))
    pairedlist(Cons(x2,Null))  ok
    pairedlist(Cons(x2,Cons(x1,x4)))  ok
      ! paired(x2,x1)=True::pairedlist(x4) !
      ! paired(x2,x1)=False::False !


            -----------------------------------------
            >>> Display of pattern tree of alter  <<<
            -----------------------------------------

alter(x1)
  alter(Null)  ok
  alter(Cons(x2,x3))
    alter(Cons(x2,Null))  ok
    alter(Cons(x2,Cons(x1,x4)))  ok
      ! paired(x2,x1)=True::alter(Cons(x1,x4)) !
      ! paired(x2,x1)=False::False !

            -------------------------------------------
            >>> Display of pattern tree of shuffle  <<<
            -------------------------------------------


shuffle(x1,x2,x3)
  shuffle(Null,x2,x3)
    shuffle(Null,Null,x3)
      shuffle(Null,Null,Null)  ok
      shuffle(Null,Null,Cons(x1,x2))  ok
    shuffle(Null,Cons(x1,x4),x3)
```

```
        shuffle(Null,Cons(x1,x4),Null)  ok
        shuffle(Null,Cons(x1,x4),Cons(x2,x5))  ok
          ! paired(x1,x2)=True::False !
          ! paired(x1,x2)=False::shuffle(Null,x4,x5) !
    shuffle(Cons(x4,x5),x2,x3)
      shuffle(Cons(x4,x5),Null,x3)
        shuffle(Cons(x4,x5),Null,Null)  ok
        shuffle(Cons(x4,x5),Null,Cons(x1,x2))  ok
          ! paired(x4,x1)=True::False !
          ! paired(x4,x1)=False::shuffle(x5,Null,x2) !
      shuffle(Cons(x4,x5),Cons(x1,x6),x3)
        shuffle(Cons(x4,x5),Cons(x1,x6),Null)  ok
        shuffle(Cons(x4,x5),Cons(x1,x6),Cons(x2,x7))  ok
          ! paired(x4,x2)=False ^ shuffle(x5,Cons(x1,x6),x7)=True::True !
          ! paired(x1,x2)=False ^ shuffle(Cons(x4,x5),x6,x7)=True::True !
          ! paired(x4,x2)=True ^ paired(x1,x2)=True::False !
          ! shuffle(x5,Cons(x1,x6),x7)=False ^ shuffle(Cons(x4,x5),x6,x7)=False::False !
          ! paired(x4,x2)=True ^ shuffle(Cons(x4,x5),x6,x7)=False::False !
          ! paired(x1,x2)=True ^ shuffle(x5,Cons(x1,x6),x7)=False::False !


--> every defined symbol is completely defined.
```

C.2 Check of ground convergence of the specification

Convergent systems of equations have the property that two terms are equal if and only if they simplify to identical ones. To get convergence, SPIKE uses the saturation technique [Kounalis Rusinowitch 87] which is a natural extension of Knuth and Bendix to conditional theories. This technique is based on a set of inference rules which is refutationally complete for first order logic with equality.

Fragment of the convergence proof is shown. Note that no completion is needed. The initial axioms are already ground convergent.

```
...

(13) opposite(Null,x1) = False

(14) opposite(x1,Null) = False

(28) paired(x1,x2) = True => shuffle(Cons(x1,x3),Null,Cons(x2,x4)) = False

(29) paired(x1,x2) = False => shuffle(Cons(x1,x3),Null,Cons(x2,x4)) = shuffle(x3,Null,x4)

(38) True = False =>

...

Right superposition of (13) into (14)  at the position [1]:

(39) False = False

Delete  (39)

...

Right superposition of (28) into (29)  at the position [1]:

(43) paired(x5,x7) = False, paired(x5,x7) = True => shuffle(x6,Null,x8) = False

Auto simplification of (43):

(43): paired(x5,x7) = True, True = False => shuffle(x6,Null,x8) = False

Delete (43): (38) proper subsumes (43)

...

The set of axioms is saturated.

-> R is ground convergent.
```

C.3 Computing test sets for R

Test sets are special induction schemes that ensure that any false boolean conjecture will be detected in finite time, given a convergent specification with completely defined functions.

The output of the SPIKE procedure that computes test sets is:

```
test set of R:

 -> bool = {False ; True}
 -> card = {B ; R}
 -> list = {Null ; Cons(B,Null) ; Cons(R,Null) ; Cons(B,Cons(B,x2)) ;
           Cons(B,Cons(R,x2)) ; Cons(R,Cons(B,x2)) ; Cons(R,Cons(R,x2))}
```

A proof has also been obtained using the following cover sets for the sort list

$$\{Null,\ Cons(x, Null),\ Cons(x, Cons(y, z))\}$$

C.4 Computing inductive positions of functions

To prove a theorem, it is sufficient to apply induction schemes to variables that occur only at special positions in a term $f(x_1, \ldots, x_n)$. This positions are called inductive positions of f.

The output of the SPIKE procedure that compute inductive positions of functions is:

```
induction positions of functions:

 -> neg: [[1]]
 -> paired: [[1];[2]]
 -> shuffle: [[1];[2];[3]]
 -> append: [[1]]
 -> rotate: [[1]]
 -> even: [[1];[1;2]]
 -> opposite: [[1];[2]]
 -> pairedlist: [[1];[1;2]]
 -> alter: [[1];[1;2]]
```

D Lemmas

We provide the system with 5 easy lemmas to help the proof.

Lemma 1 *Let* $x1$ *and* $x2$ *two lists of cards. If* $x1$ *is even, then* $append(x1, cons(x2, x3))$ *is even iff* $cons(x2, x3)$ *is even.*

```
even(x1) = True  => even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3))
```

Lemma 2 *Let* $x1$ *and* $x2$ *two lists of cards. If* $x1$ *is odd, then* $append(x1, cons(x2, x3))$ *is even iff* $x3$ *is even.*

```
even(x1) = False => even(append(x1,Cons(x2,x3))) = even(x3)
```

Lemma 3 *If* $Cons(x1, y1)$ *is a list of cards whose colors do not alternate then* $Cons(x1, append(y1, y2))$ *is a list of cards whose colors do not alternate either.*

```
alter(Cons(x1,x2)) = False => alter(Cons(x1,append(x2,x3))) = False
```

Lemma 4 *If* $Cons(x1, y1)$ *is a list of cards whose colors alternate and* $y1$ *is of odd length, then* $Cons(x1, append(y1, y2))$ *alternate iff* $Cons(neg(x1), y2)$ *alternate.*

```
alter(Cons(x1,x2)) = True, even(x2) = False => alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3))
```

Lemma 5 *If* $Cons(x1, y1)$ *is a list of cards whose colors alternate and* $y1$ *is of even length, then* $Cons(x1, append(y1, y2))$ *alternate iff* $Cons(x1, y2)$ *alternate.*

```
alter(Cons(x1,x2)) = True, even(x2) = True  => alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3))
```

Remark

If we formulate the problem as Hantao Zhang in RRL that is with the help of a *conditional function "cond"*, then two lemmas are sufficient for SPIKE to prove Gilbreath card trick, namely:

```
1. even(append(x1,Cons(x2,x3))) == cond(even(x1) = True, even(Cons(x2,x3)), even(x3))

2. alter(Cons(x1,append(x2,x3))) ==
   cond(alter(Cons(x1,x2))=False, False,
   cond(even(x2)=True, alter(Cons(x1,x3)), alter(Cons(neg(x1),x3))))
```

Proof of lemmas

During a preliminary attempt of a proof, we have noticed that the pattern $even(append(x1, Cons(x2, x3)))$ was a cause of divergence. This has motivated us to introduce lemmas 1 and 2. They allow to replace $even(append(x1, Cons(x2, x3)))$ by $even(Cons(x2, x3))$ if $x1$ is *even*, and $even(x3)$ if $x1$ is *odd*. In the same way, we eliminate *alter(Cons(x1,append(x2,x3)))*, thanks to lemmas 3, 4 and 5.

The 5 lemmas are proved in a single run. Below, we show partial transcripts of the proof session:

```
E0 = {alter(Cons(x1,x2)) = True, even(x2) = False => alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)) ;
      alter(Cons(x1,x2)) = True, even(x2) = True  => alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)) ;
      alter(Cons(x1,x2)) = False => alter(Cons(x1,append(x2,x3))) = False ;
      even(x1) = True  => even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)) ;
      even(x1) = False => even(append(x1,Cons(x2,x3))) = even(x3)}
```

Since in our frameworks an atom is considered to be simpler than its negation, we simplify clauses into positive ones.

```
Simplification of:
    alter(Cons(x1,x2)) = True, even(x2) = False => alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3))
       alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)), alter(Cons(x1,x2)) = False, even(x2) = True

Simplification of:
    alter(Cons(x1,x2)) = True, even(x2) = True => alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3))
       alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False

Simplification of:
    alter(Cons(x1,x2)) = False => alter(Cons(x1,append(x2,x3))) = False
       alter(Cons(x1,append(x2,x3))) = False, alter(Cons(x1,x2)) = True

Simplification of:
    even(x1) = True => even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3))
       even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)), even(x1) = False

Simplification of:
    even(x1) = False => even(append(x1,Cons(x2,x3))) = even(x3)
       even(append(x1,Cons(x2,x3))) = even(x3), even(x1) = True

E1 = {alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)), alter(Cons(x1,x2)) = False, even(x2) = True ;
      alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False ;
      alter(Cons(x1,append(x2,x3))) = False, alter(Cons(x1,x2)) = True ;
      even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)), even(x1) = False ;
      even(append(x1,Cons(x2,x3))) = even(x3), even(x1) = True}

H1 = {}
```

To prove *alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)), alter(Cons(x1,x2)) = False, even(x2) = True* the induction will be done on *x1*, and will follow the scheme:

```
Application of generate on:
    alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)), alter(Cons(x1,x2)) = False, even(x2) = True:
 1) alter(Cons(B,x2)) = False, even(x2) = True, alter(Cons(B,append(x2,x3))) = alter(Cons(R,x3)) ;
 2) alter(Cons(R,x2)) = False, even(x2) = True, alter(Cons(R,append(x2,x3))) = alter(Cons(B,x3))


E2 = {alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False ;
      alter(Cons(x1,append(x2,x3))) = False, alter(Cons(x1,x2)) = True ;
      even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)), even(x1) = False ;
      even(append(x1,Cons(x2,x3))) = even(x3), even(x1) = True ;
      alter(Cons(B,x1)) = False, even(x1) = True, alter(Cons(B,append(x1,x2))) = alter(Cons(R,x2)) ;
      alter(Cons(R,x1)) = False, even(x1) = True, alter(Cons(R,append(x1,x2))) = alter(Cons(B,x2))}

H2 = {alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)), alter(Cons(x1,x2)) = False, even(x2) = True}
```

To prove *alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False* the induction will be done on *x2*, and will follow the scheme:

```
Application of generate on:
    alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False:
 1) alter(Cons(x1,Null)) = False, even(Null) = False, alter(Cons(x1,x3)) = alter(Cons(x1,x3)) ;
 2) alter(Cons(x1,Cons(B,Null))) = False,
    even(Cons(B,Null)) = False, alter(Cons(x1,Cons(B,x3))) = alter(Cons(x1,x3)) ;
 3) alter(Cons(x1,Cons(R,Null))) = False, even(Cons(R,Null)) = False,
    alter(Cons(x1,Cons(R,x3))) = alter(Cons(x1,x3)) ;
 4) alter(Cons(x1,Cons(B,Cons(B,x2)))) = False,
    even(Cons(B,Cons(B,x2))) = False, alter(Cons(x1,Cons(B,Cons(B,append(x2,x3))))) = alter(Cons(x1,x3)) ;
 5) alter(Cons(x1,Cons(B,Cons(R,x2)))) = False,
    even(Cons(B,Cons(R,x2))) = False, alter(Cons(x1,Cons(B,Cons(R,append(x2,x3))))) = alter(Cons(x1,x3)) ;
 6) alter(Cons(x1,Cons(R,Cons(B,x2)))) = False,
    even(Cons(R,Cons(B,x2))) = False, alter(Cons(x1,Cons(R,Cons(B,append(x2,x3))))) = alter(Cons(x1,x3)) ;
 7) alter(Cons(x1,Cons(R,Cons(R,x2)))) = False,
    even(Cons(R,Cons(R,x2))) = False, alter(Cons(x1,Cons(R,Cons(R,append(x2,x3))))) = alter(Cons(x1,x3))

...
```

The proof of $alter(Cons(x1, Cons(B, Cons(B, x2)))) = False$, $even(x2) = False$, $alter(Cons(x1, Cons(B, Cons(B, append(x2, x3))))) = alter(Cons(x1, x3))$ is done by a case-analysis on whether $paired(x1, B) = True$ or $paired(x1, B) = False$.

```
Application of case rewriting using R on:
    alter(Cons(x1,Cons(B,Cons(B,x2)))) = False, even(x2) = False,
         alter(Cons(x1,Cons(B,Cons(B,append(x2,x3))))) = alter(Cons(x1,x3)):
 1) paired(x1,B) = True => even(x2) = False,
    alter(Cons(x1,Cons(B,Cons(B,append(x2,x3))))) = alter(Cons(x1,x3)), alter(Cons(B,Cons(B,x2))) = False ;
 2) paired(x1,B) = False => even(x2) = False,
    alter(Cons(x1,Cons(B,Cons(B,append(x2,x3))))) = alter(Cons(x1,x3)), False = False ;
 ...

E60 = {even(x1) = False, alter(Cons(B,append(x1,x2))) = alter(Cons(B,x2)),
       alter(Cons(B,x1)) = False, True = False}

H60 = {alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False ;
       ...}

Delete  even(x1) = False, alter(Cons(B,append(x1,x2))) = alter(Cons(B,x2)),
          alter(Cons(B,x1)) = False, True = False
 it is subsumed by:
   alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)),    alter(Cons(x1,x2)) = False,
        even(x2) = False of H60

E61 = {}

The initial conjectures are inductive consequences of R
```

The following sub lemmas have been generated automatically during the proof and have played a role in it:

```
{even(append(x1,Cons(x2,x3))) = even(x3), even(x1) = True ;
 alter(Cons(x1,append(x2,x3))) = False, alter(Cons(x1,x2)) = True ;
 even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)), even(x1) = False ;
 alter(Cons(B,x1)) = False, even(x1) = True, alter(Cons(B,append(x1,x2))) = alter(Cons(R,x2)) ;
 alter(Cons(R,x1)) = False, even(x1) = True, alter(Cons(R,append(x1,x2))) = alter(Cons(B,x2)) ;
 alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)), alter(Cons(x1,x2)) = False, even(x2) = False}
```

E The main theorem

Below, we show partial transcripts of the proof session, to illustrate the main steps of the proof:

```
E0 = {alter(append(x1,x2)) = True, even(append(x1,x2)) = True,
          opposite(x1,x2) = True, shuffle(x1,x2,x3) = True => pairedlist(x3) = True ;
     alter(append(x1,x2)) = True, even(append(x1,x2)) = True,
          opposite(x1,x2) = False, shuffle(x1,x2,x3) = True => pairedlist(rotate(x3)) = True}

L = {alter(Cons(x1,x2)) = True, even(x2) = False => alter(Cons(x1,append(x2,x3))) = alter(Cons(neg(x1),x3)) ;
     alter(Cons(x1,x2)) = True, even(x2) = True => alter(Cons(x1,append(x2,x3))) = alter(Cons(x1,x3)) ;
     alter(Cons(x1,x2)) = False => alter(Cons(x1,append(x2,x3))) = False ;
     even(x1) = True => even(append(x1,Cons(x2,x3))) = even(Cons(x2,x3)) ;
     even(x1) = False => even(append(x1,Cons(x2,x3))) = even(x3)}

Simplification of:
    alter(append(x1,x2)) = True, even(append(x1,x2)) = True,
         opposite(x1,x2) = True, shuffle(x1,x2,x3) = True => pairedlist(x3) = True
    pairedlist(x3) = True, alter(append(x1,x2)) = False,
         even(append(x1,x2)) = False, opposite(x1,x2) = False, shuffle(x1,x2,x3) = False

Simplification of:
    alter(append(x1,x2)) = True, even(append(x1,x2)) = True,
         opposite(x1,x2) = False, shuffle(x1,x2,x3) = True => pairedlist(rotate(x3)) = True
    pairedlist(rotate(x3)) = True, alter(append(x1,x2)) = False,
         even(append(x1,x2)) = False, opposite(x1,x2) = True, shuffle(x1,x2,x3) = False

E1 = {pairedlist(x1) = True, alter(append(x2,x3)) = False,
           even(append(x2,x3)) = False, opposite(x2,x3) = False, shuffle(x2,x3,x1) = False ;
      pairedlist(rotate(x1)) = True, alter(append(x2,x3)) = False,
           even(append(x2,x3)) = False, opposite(x2,x3) = True, shuffle(x2,x3,x1) = False}

H1 = {}
```

The next case-rewriting step eliminate *alter(Cons(B,append(x1,Cons(B,Cons(B,x2)))))*. Several clauses are generated by this inference step. Some of them are shown below:

```
Application of case rewriting using L on:
    alter(Cons(B,append(x1,Cons(B,Cons(B,x2))))) = False,
         even(append(x1,Cons(B,Cons(B,x2)))) = False,
             shuffle(Cons(R,Cons(B,x1)),Cons(B,Cons(B,x2)),Cons(B,Cons(B,x3))) = False:

 1) alter(Cons(B,x1)) = True, even(x1) = False => even(append(x1,Cons(B,Cons(B,x2)))) = False,
    shuffle(Cons(R,Cons(B,x1)),Cons(B,Cons(B,x2)),Cons(B,Cons(B,x3))) = False,
    alter(Cons(neg(B),Cons(B,Cons(B,x2)))) = False ;
 2) alter(Cons(B,x1)) = True, even(x1) = True => even(append(x1,Cons(B,Cons(B,x2)))) = False,
    shuffle(Cons(R,Cons(B,x1)),Cons(B,Cons(B,x2)),Cons(B,Cons(B,x3))) = False,
    alter(Cons(B,Cons(B,Cons(B,x2)))) = False ;
 3) alter(Cons(B,x1)) = False => even(append(x1,Cons(B,Cons(B,x2)))) = False,
    shuffle(Cons(R,Cons(B,x1)),Cons(B,Cons(B,x2)),Cons(B,Cons(B,x3))) = False, False = False
```

...

Application of inductive rewriting to eliminate *even(append(x1,Cons(R,Cons(B,x2))))*.

```
E191 = {even(append(x1,Cons(R,Cons(B,x2)))) = False,
        shuffle(Cons(R,Cons(B,x1)),Cons(R,Cons(B,x2)),Cons(R,Cons(B,x3))) = False,
         pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True, alter(Cons(neg(B),Cons(R,Cons(B,x2)))) = False,
                    alter(Cons(B,x1)) = False, even(x1) = True ;
        even(append(x1,Cons(R,Cons(B,x2)))) = False,
            shuffle(Cons(R,Cons(B,x1)),Cons(R,Cons(B,x2)),Cons(R,Cons(B,x3))) = False,
             pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True, alter(Cons(B,Cons(R,Cons(B,x2)))) = False,
                     alter(Cons(B,x1)) = False, even(x1) = False ;
        ...}

H191 = {pairedlist(rotate(x1)) = True, alter(append(x2,x3)) = False,
              even(append(x2,x3)) = False, opposite(x2,x3) = True, shuffle(x2,x3,x1) = False ;
       pairedlist(x1) = True, alter(append(x2,x3)) = False,
             even(append(x2,x3)) = False, opposite(x2,x3) = False, shuffle(x2,x3,x1) = False}

Simplification of:
    even(x1) = False => even(append(x1,Cons(R,Cons(B,x2)))) = False,
        shuffle(Cons(R,Cons(B,x1)),Cons(R,Cons(B,x2)),Cons(R,Cons(B,x3))) = False,
         pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True, alter(Cons(neg(B),Cons(R,Cons(B,x2)))) = False,
          alter(Cons(B,x1)) = False by R U L[H191 U E191]:

    even(x1) = False => even(Cons(B,x2)) = False,
        shuffle(Cons(R,Cons(B,x1)),Cons(R,Cons(B,x2)),Cons(R,Cons(B,x3))) = False,
        pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True, False = False, alter(Cons(B,x1)) = False

Simplification of:
    even(x1) = True => even(append(x1,Cons(R,Cons(B,x2)))) = False, shuffle(Cons(R,Cons(B,x1)),
              Cons(R,Cons(B,x2)),Cons(R,Cons(B,x3))) = False,
              pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True,
              alter(Cons(B,Cons(R,Cons(B,x2)))) = False, alter(Cons(B,x1)) = False by R U L[H191 U E191]:


    even(x1) = True => even(x2) = False, shuffle(Cons(R,Cons(B,x1)),Cons(R,Cons(B,x2)),
         Cons(R,Cons(B,x3))) = False, pairedlist(Cons(B,append(x3,Cons(R,Null)))) = True,
         alter(Cons(B,x2)) = False, alter(Cons(B,x1)) = False

...

..
```

Application of a case rewriting to simplify *shuffle(Cons(R,Null),Cons(B,Null),Cons(R,Cons(B,x1)))*:

```
Application of case rewriting using R on:
    shuffle(Cons(R,Null),Cons(B,Null),Cons(R,Cons(B,x1))) = False, pairedlist(x1) = True:
 1) paired(R,R) = False, shuffle(Null,Cons(B,Null),Cons(B,x1)) = True => pairedlist(x1) = True, True = False ;
 2) paired(B,R) = False, shuffle(Cons(R,Null),Null,Cons(B,x1)) = True => pairedlist(x1) = True, True = False ;
 3) paired(R,R) = True, paired(B,R) = True => pairedlist(x1) = True, False = False ;
 4) shuffle(Null,Cons(B,Null),Cons(B,x1)) = False,
    shuffle(Cons(R,Null),Null,Cons(B,x1)) = False => pairedlist(x1) = True, False = False ;
 5) paired(R,R) = True, shuffle(Cons(R,Null),Null,Cons(B,x1)) = False => pairedlist(x1) = True, False = False ;
 6) paired(B,R) = True, shuffle(Null,Cons(B,Null),Cons(B,x1)) = False => pairedlist(x1) = True, False = False


...

E912 = {}

The initial conjectures are inductive consequences of R
```

37 lemmas were generated automatically to prove the conjectures, for example:

```
pairedlist(x1) = True, shuffle(Null,Null,x1) = False ;

pairedlist(x1) = True, alter(Cons(R,x2)) = False, even(x2) = True, shuffle(x2,Cons(R,Null),x1) = False ;

alter(Cons(R,x1)) = False, even(x1) = False, shuffle(Null,x1,x2) = False,
     pairedlist(Cons(R,append(x2,Cons(B,Null)))) = True ;

even(Cons(B,x1)) = False, pairedlist(x2) = True, alter(Cons(B,x1)) = False,
    alter(Cons(R,x3)) = False, even(x3) = True, shuffle(Cons(R,x3),Cons(B,x1),x2) = False ;
```

F Refutation of conjectures

R is a boolean ground convergent rewrite system with completely defined functions. Under these hypotheses SPIKE can refute any false boolean conjecture in finite time. Here is an example of a refutation of a false conjecture.

```
E0 = {alter(append(x1,x2)) = True, even(append(x1,x2)) = True,
           opposite(x1,x2) = True, shuffle(x1,x2,x3) = True => pairedlist(x3) = False}

Simplification of:
    alter(append(x1,x2)) = True, even(append(x1,x2)) = True, opposite(x1,x2) = True,
         shuffle(x1,x2,x3) = True  => pairedlist(x3) = False
       pairedlist(x3) = False, alter(append(x1,x2)) = False,
            even(append(x1,x2)) = False, opposite(x1,x2) = False, shuffle(x1,x2,x3) = False

E1 = {pairedlist(x1) = False, alter(append(x2,x3)) = False,
           even(append(x2,x3)) = False, opposite(x2,x3) = False, shuffle(x2,x3,x1) = False}

H1 = {}

...


E144 = {pairedlist(x1) = False, shuffle(Null,Null,x1) = False ;
        ...}


H144 = {pairedlist(x1) = False, alter(append(x2,x3)) = False,
           even(append(x2,x3)) = False, opposite(x2,x3) = False, shuffle(x2,x3,x1) = False}
...

Application of generate on:
    pairedlist(x1) = False, shuffle(Null,Null,x1) = False:
 1) shuffle(Null,Null,Null) = False, True = False ;
 2) shuffle(Null,Null,Cons(B,Null)) = False, True = False ;
 3) shuffle(Null,Null,Cons(R,Null)) = False, True = False ;
 4) shuffle(Null,Null,Cons(B,Cons(B,x1))) = False, False = False ;
 5) shuffle(Null,Null,Cons(B,Cons(R,x1))) = False, pairedlist(x1) = False ;
 6) shuffle(Null,Null,Cons(R,Cons(B,x1))) = False, pairedlist(x1) = False ;
 7) shuffle(Null,Null,Cons(R,Cons(R,x1))) = False, False = False


Simplification of:
    shuffle(Null,Null,Null) = False, True = False by R U L[H146 U E146]:
    True = False
```

The quasi-inconsistant clause $True = False$ is generated.

```
E149 = {True = False}

H149 = {pairedlist(x1) = False, shuffle(Null,Null,x1) = False ;
       pairedlist(x1) = False, alter(append(x2,x3)) = False,
             even(append(x2,x3)) = False, opposite(x2,x3) = False, shuffle(x2,x3,x1) = False}
```

Hence:

```
The initial conjecture is not an inductive consequence of R.
```

Appendix 2: Proofs

Here we prove the lemmas 2 and 4 for the inference system N:

Proof of lemma 2: Let $\Im = min_{\prec_c}\{C\sigma / C \in \cup_i E_i$ and there is a ground substitution σ irreducible by R such that $R \not\models_{ind} C\sigma\}$. $\Im \neq \emptyset$ since $R \not\models_{ind} E_0$ and $\prec_c$ is well-founded. Consider a clause $C \equiv \wedge_k\ a_k = b_k \Rightarrow \vee_l\ c_l = d_l$ which is minimal in $\Im$ with respect to the subsumption ordering. It is sufficient to prove that C cannot be simplified nor deleted, and that neither *generate* nor *complement* can be applied to C; this shows that *fail* applies since the clause C must not persist in the derivation by the fairness hypothesis.

Hence let us assume that $C \in E_j$ and $(E_j, H_j) \vdash_N (E_{j+1}, H_{j+1})$ by some rule applied to C. We discuss now the situation according to which rule is applied. In every case we shall derive a contradiction. In order to simplify the notations we write E for E_j, H for H_j and P for $C\sigma$.

Since σ is a ground substitution that is irreducible by R, there exists a test substitution σ_0 of C and a ground substitution θ such that $\sigma = \sigma_0\theta$. Before proving the lemma we show the following claim:

claim 1

1.1 If there is i_0 such that $C[a_{i_0}] \rightsquigarrow_{R[H \cup E];r} C'$ where r is the subclause of C such that $C \equiv \neg(a_{i_0} = b_{i_0}) \vee r$, then $R \not\models_{ind} C'\sigma$.

1.2 If there is i_0 such that $C\sigma_0[a_{i_0}\sigma_0] \rightsquigarrow_{R[H \cup E];r\sigma_0} C'$ where r is the subclause of C such that $C \equiv \neg(a_{i_0} = b_{i_0}) \vee r$, then $R \not\models_{ind} C'\theta$.

1.3 The application of *case_rewriting* to C generates a clause C_k such that $R \not\models_{ind} C_k\sigma$ and $C_k\sigma \prec C\sigma$.

1.4 The application of *case_rewriting* to $C\sigma_0$ generates a clause C_k such that $R \not\models_{ind} C_k\theta$ and $C_k\theta \prec C\sigma$.

proof: We prove only 1.1 and 1.3 since the proof of 1.2 and 1.4 are similar respectively.

1.1 If there exists i_0 such that $a_{i_0} \rightsquigarrow_{R[H \cup E];r} a'_{i_0}$, we also have by lemma 1, $a_{i_0}\sigma \rightsquigarrow_{R[H \cup E];r\sigma} a'_{i_0}\sigma$. There are two cases to consider:
If $a_{i_0}\sigma \rightarrow_{prem(r\sigma)} a'_{i_0}\sigma$, then $R \models a_{i_0}\sigma = a'_{i_0}\sigma$ since $R \not\models_{ind} C\sigma$.
Otherwise, there exists $D \equiv \wedge_i\ a'_i = b'_i \Rightarrow c' \rightarrow d' \ \in\ R$ such that: $a_{i_0}\sigma \equiv a_{i_0}\sigma[c'\tau]$, $a'_{i_0}\sigma \equiv a_{i_0}\sigma[d'\tau]$ and:

- **i.** for all i there exist e'_i, e''_i such that: $a'_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e'_i$, $b'_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e''_i$ and $e'_i =_{prem(r\sigma)} e''_i$.
- **ii.** $\{a_{i_0}\sigma\} \gg \{a'_1\tau,\ b'_1\tau,\ \ldots\ , a'_n\tau,\ b'_n\tau\}$.

Let us show that for all i, $R \models a'_i\tau = b'_i\tau$:

We first prove that $a'_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e'_i$ implies $R \models a'_i\tau = e'_i$ by induction on $a'_i\tau$ w.r.t. $\succ$. We have $a'_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e'_i$. Let $c[g\lambda] \mapsto c[h\lambda]$ be one of the proof step of this derivation.
If $c[g\lambda] \rightarrow_{prem(r\sigma)} c[h\lambda]$, we have $R \models c[g\lambda] = c[h\lambda]$.
Otherwise, suppose there is $Q \equiv \wedge_i\ g_i = h_i \Rightarrow g = h \ \in R \cup H \cup E$ such that:

1. $g\lambda \succ h\lambda$.
2. $\forall i\ \exists f'_i,\ f''_i$ such that $g_i\lambda \mapsto^*_{R \cup H \cup E;r\sigma} f'_i$, $h_i\lambda \mapsto^*_{R \cup H \cup E;r\sigma} f''_i$ and $f'_i =_{prem(r\sigma)} f''_i$.
3. $c[g\lambda] \succ \{g_1\lambda,\ h_1\lambda \ldots, g_n\lambda,\ h_n\lambda\}$.

We have: $g_i\lambda \prec c[g\lambda] \preceq a'_i\tau$, Hence by induction hypothesis we deduce that $R \models g_i\lambda = f'_i$. In the same way $R \models h_i\lambda = f''_i$. On the other side $R \models f'_i = f''_i$ (since $R \not\models_{ind} C\sigma$). This implies that for all i, $R \models g_i\lambda = h_i\lambda$. On the other hand we have : $g_i\lambda \prec a'_i\tau \prec a_{i_0}\sigma$, $h_i\lambda \prec a_{i_0}\sigma$,

$g\lambda \preceq a'_i\tau \prec a_{i_0}\sigma$ and $h\lambda \prec g\lambda \prec a_{i_0}\sigma$. Hence $Q\lambda \prec_c P$ and by hypothesis $Q\lambda$ is valid in R. We therefore have $R \models c[g\lambda] = c[h\lambda]$ and as a consequence $R \models a'_i\tau = e'_i$.

In the same vein we have $R \models b'_i\tau = e''_i$. On the other side $R \models e'_i = e''_i$. From the previous results we deduce that for all i, $R \models a'_i\tau = b'_i\tau$.

Since $D \in R$ we have $R \models c'\tau = d'\tau$. This implies that $R \models a'_{i_0}\sigma = b_{i_0}\sigma$ since $R \models a_{i_0}\sigma = b_{i_0}\sigma$ ($R \not\models_{ind} C\sigma$ by hypothesis) and $R \models a_{i_0}\sigma = a'_{i_0}\sigma$ (see above)

Therefore we have $R \not\models_{ind} (\wedge_{i \neq i_0}\ a_i = b_i \wedge a'_{i_0} = b_{i_0} \Rightarrow \vee_j c_j = d_j)\sigma$. This shows a contradiction since it proves that we can find an instance of a clause in E_{j+1} which is not valid and smaller than $C\sigma$.

1.3 Assume that the rule *case_rewriting* can be applied to C. Then we have: $C \equiv \neg(a_k[s_1\lambda_1]_{u_1} = b_k) \vee r \equiv \ldots \equiv \neg(a_k[s_l\lambda_l]_{u_l} = b_k) \vee r$ with $\{P_1 \Rightarrow s_1 \rightarrow t_1,\ \ldots,\ P_l \Rightarrow s_l \rightarrow t_l\} \subseteq R$. The result is: $\{C_1 \equiv \neg(a_k[t_1\lambda_1]_{u_1} = b_k) \vee \neg P_1\lambda_1 \vee r, \ldots, C_l \equiv \neg(a_k[t_l\lambda_l]_{u_l} = b_k) \vee \neg P_l\lambda_l \vee r\} \cup CNF(P_1\lambda_1 \vee \ldots \vee P_l\lambda_l)$ with $P_i\lambda_i \ll \{s_i\lambda_i\}$ for all i. Hence any clause in $CNF(P_1\lambda_1 \vee\ \ldots\ \vee P_l\lambda_l)\sigma$ is $\ll \{a_k\sigma\}$. This implies that any clause in $CNF(P_1\lambda_1 \vee\ \ldots\ \vee P_l\lambda_l)\sigma$ is $\prec_c P$. As a consequence there exists $i \in [1,..,l]$ such that $R \models_{ind}\ P_i\lambda_i\sigma$. Let C_i be $\neg(a_k\sigma[t_i\lambda_i]_{u_i} = b_k) \vee \neg P_i\lambda_i \vee r$. We have $P_i\lambda_i\sigma \ll \{s_i\lambda_i\sigma\}$ and then $P_i\lambda_i\sigma \ll \{a_k\sigma[s_i\lambda_i\sigma]\}$. Since $t_i\lambda_i \prec s_i\lambda_i$ we also have $a_k\sigma[t_i\lambda_i\sigma] \prec a_k\sigma[s_i\lambda_i\sigma]$. Finally, $C_i \prec_c P$. On one hand $R \not\models_{ind} C\sigma$ and therefore $R \not\models_{ind} r\sigma$. On the other hand $R \models_{ind} (P_i \Rightarrow s_i \rightarrow t_i)\lambda_i\sigma$, $R \models_{ind} P_i\lambda_i\sigma$ and therefore $R \models s_i\lambda_i\sigma = t_i\lambda_i\sigma$ and also $R \models a_k\sigma[t_i\lambda_i\sigma]_{u_i} = b_k\sigma$. Putting everything together we get $R \not\models_{ind} C_i\sigma$. A contradiction raises from the generation of a non valid (instance of a) clause smaller than $C\sigma$. The claim is proved.

We show now that whatever rule is applied to C, we obtain a contradiction:
generate: There are two cases:

A. *generate* applies to $C \equiv \neg(a_k = b_k) \vee r$. Note that $C\sigma$ cannot be a tautology. We have again two possibilities:

a. there exists a_0 such that $a_k\sigma_0 \mapsto_{R[H \cup E];r\sigma_0} a'_0$. Let $Q \equiv \neg(a'_0 = b_k\sigma_0) \vee r\sigma_0$, $Q \in \cup_i E_i$. From Claim 1.2, $R \not\models_{ind} Q\theta$. On the other hand $Q\theta \prec_c P$ since $a'_0\theta = b_k\sigma \prec_e a_k\sigma = b_k\sigma$, which is absurd.

b. If *case_rewriting* applies to $C\sigma_0$, then from Claim 1.4, there is a clause $C' \in \cup_i E_i$ with $R \not\models_{ind} C'\theta$ and $C'\theta \prec_c P$. This is also absurd.

B. *generate* applies to $C \equiv c_k = d_k \vee r'$, This case is similar to A.

case simplify Since *case simplification* applies to C then by Claim 1.3 there is a clause $C' \in \cup_i E_i$ with $R \not\models_{ind} C'\sigma$ and $C'\sigma \prec_c P$, contradiction. The proof is similar whether we use R or L for case simplification.

simplify: If *simplify* applies to C, there are two cases:

A. *Simplify* applies to $C \equiv \neg(a_k = b_k) \vee r$. there are again two possibilities:

A1. $a_k \mapsto_{R[H \cup E];r} a'$. Let $C' \equiv \neg(a' = b_k) \vee r$. By Claim 1.1 $R \not\models_{ind} C'\sigma$ and on the other hand $C'\sigma \prec_c P$, contradiction.

A2. $a_k \rightsquigarrow_{H \cup E[R];r} a'$, we also have $a_k\sigma \rightsquigarrow_{H \cup E[R];r\sigma} a'\sigma$. There are two cases:
If $a_k\sigma \rightarrow_{prem(r\sigma)} a'\sigma$, then $R \models a_k\sigma = a'\sigma$ since $R \not\models_{ind} C\sigma$.
Otherwise, there exists $D \equiv \wedge_i\ s_i = t_i \Rightarrow s = t\ \in\ H \cup E$ such that: $a_k\sigma \equiv a_k\sigma[s\tau]$ and $a'\sigma \equiv a_k\sigma[t\tau]$. Let C' be $\neg(a' = b_k) \vee r$. Then:

i. either $R \models s\tau = t\tau$. Then $R \not\models_{ind} C'\sigma$ and $C'\sigma \prec P$, contradiction.

ii. or $R \not\models s\tau = t\tau$

Let us show that $R \not\models_{ind} C_s\tau$. We have:

- $\{a_k\sigma\} \gg \{s_1\tau,\ t_1\tau,\ \dots\ , s_n\tau,\ t_n\tau\}$.
- $\forall i\ \ \exists e'_i,\ e''_i$ *such that* : $s_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e'_i$, $t_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e''_i$ et $e'_i =_{prem(r\sigma)} e''_i$.

Let us prove that for all i $R \models s_i\tau = t_i\tau$: We begin by proving that $R \models s_i\tau\sigma = e'_i$ by induction on $s_i\tau\sigma$ w.r.t. $\succ$: We have $s_i\tau \mapsto^*_{R \cup H \cup E;r\sigma} e'_i$. Let $c[g\lambda] \mapsto c[h\lambda]$ be one of the proof step of this derivation.

If $c[g\lambda] \rightarrow_{prem(r\sigma)} c[h\lambda]$, we have $R \models c[g\lambda] = c[h\lambda]$.

Otherwise, suppose there is $Q \equiv \wedge_i\ g_i = h_i \Rightarrow g = h\ \ \in H \cup E$ such that :

1. $g\lambda \succ h\lambda$.
2. $\forall i\ \exists f'_i,\ f''_i$ such that $g_i\lambda \mapsto^*_{R \cup H \cup E;r\sigma} f'_i$, $h_i\lambda \mapsto^*_{R \cup H \cup E;r\sigma} f''_i$ et $f'_i =_{prem(r\sigma)} f''_i$.
3. $c[g\lambda] \succ \{g_1\lambda,\ , h_1\lambda \dots, g_n\lambda,\ h_n\lambda\}$.

We have: $g_i\lambda \prec c[g\lambda] \preceq s_i\tau$ and, by induction hypothesis, also: $R \models g_i\lambda = f'_i$. Similarly $h_i\lambda \prec s_i\tau$ and therefore $R \models h_i\lambda = f''_i$. Since $R \models f'_i = f''_i$ which implies that $\forall i\ R \models g_i\lambda = h_i\lambda$. We also have: $g_i\lambda \preceq s_i\tau \prec a_k\sigma$, $h_i\lambda \prec s_i\tau \prec a_k\sigma$, $g\lambda \prec a_k\sigma$ and $h\lambda \prec a_k\sigma$. Hence $Q\lambda \prec_c P$ and $R \models c[g\lambda] = c[h\lambda]$. It follows that $R \models s_i\tau = e'_i$. In the same way we can verify that $R \models t_i\tau = e''_i$. On one hand, $R \models e'_i = e''_i$, hence $\forall i\ R \models s_i\tau = t_i\tau$, on the other hand $R \not\models s\tau = t\tau\sigma$ hence $R \not\models_{ind} C_s\tau$. This yields the contradiction since $C_s \in \cup_i E_i$ and $C_s\tau \prec_c P$.

B. the case where simplify applies to $C \equiv c_k = d_k \vee r'$ is similar to A.

right simplify of constructors: Assume that the rule *right simplify of constructors* applies to $C \equiv f(s_1, \dots, s_n) = f(t_1, \dots, t_n) \vee r$. Then we have $R \not\models f(s_1, \dots, s_n)\sigma = f(t_1, \dots, t_n)\sigma$. Since f is a free constructor there exists i such that $R \not\models s_i\sigma = t_i\sigma$. Let $Q \equiv s_i = t_i \vee r$. Note that $Q \in \cup_i E_i$, $R \not\models_{ind} Q\sigma$ and $Q\sigma \prec_c P$, contradiction.

left simplify of constructors: Assume that the rule *left simplify of constructors* applies to $C \equiv \neg(f(s_1, \dots, s_n) = f(t_1, \dots, t_n)) \vee r$. Then we have $R \models f(s_1, \dots, s_n)\sigma = f(t_1, \dots, t_n)\sigma$. Since f is a free constructor for all i, $R \models s_i\sigma = t_i\sigma$. Let $Q \equiv \vee_i \neg(s_i = t_i) \vee r$. Then $Q \in \cup_i E_i$, $R \not\models_{ind} Q\sigma$ and $Q\sigma \prec_c P$, contradiction.

deletion of a trivial clause: Since $R \not\models_{ind} C\sigma$, C is not a tautology and this rule need not be considered.

subsumption: Since $R \not\models_{ind} C\sigma$, C cannot be subsumed by an axiom from R. If there is $C' \in H \cup (E \setminus \{C\})$ such that $C \equiv C'\tau \vee r$, we have $R \not\models_{ind} C'\tau\sigma$ so $r = \emptyset$ and $\tau = \mathcal{I}$ since C is minimal in $\Im$ w.r.t. the subsumption ordering. As a consequence $C' \notin (E \setminus \{C\})$. Assume that $C' \in H$. Hence *generate* has been applied to C'. Therefore *generate* can be also applied to C in contradiction with a previous case.

complement: Assume that $C \equiv \neg(a_k\tau = b_k\tau)\ \vee\ r$. *Complement* generate $Q \equiv a_k\tau = b'\tau\ \vee\ r$. But $R \not\models (a_k\tau)\sigma = (b'\tau)\sigma$ since $\neg(b_k = b') \in R$. Hence $R \not\models_{ind} Q\sigma$. Either we have $b'\tau \prec b_k\tau$ and also $Q\sigma \prec_c P$, absurd. Or $b'\tau \approx b_k\tau$ and $Q\sigma \prec_c P$ again since $nln(Q\sigma) = nln(C\sigma) - 1$. This is also impossible.

Proof of lemma 4: Let R be a convergent rewrite system such that all defined symbols are completely defined. Let C be the clause $\wedge_i\ a_i = b_i \Rightarrow \vee_j\ c_j = d_j\ \in E_j$ and assume that $(E_j, H_j) \vdash_I (E_{j+1}, H_{j+1})$ by application of an inference rule on C. Let us show that $\forall i \leq j\ \ R \models_{ind} E_i$ implies $R \models_{ind} E_{j+1}$.

Let us first prove the following claim:

claim 2

2.1 if there exists i_0 such that $C[a_{i_0}] \rightsquigarrow_{C_s[R \cup H_j \cup E_j];r} C'$ where r is the subclause of C such that $C \equiv \neg(a_{i_0} = b_{i_0}) \ \vee \ r$ and $C_s \in R \cup H_j \cup E_j$. then $R \models_{ind} C'$.

2.2 if *case_rewriting* applies to C the derived clauses are inductive consequences of R.

proof:
2.1 If $a_{i_0} \rightsquigarrow_{C_s[R \cup H_j \cup E_j];r} a'_{i_0}$, then let C' be the clause $\neg(a'_{i_0} = b_{i_0}) \ \vee \ r$. Consider $C'\theta$ a ground instance of C'. We can assume that for all $i \neq i_0$, $R \models a_i\theta = b_i\theta$, $R \not\models a_{i_0}\theta = b_{i_0}\theta$ and for all j $R \not\models c_j\theta = d_j\theta$ (otherwise the conclusion is immediately derived). We have $a_{i_0}\theta \rightsquigarrow_{C_s[R \cup H_j \cup E_j];r\theta} a'_{i_0}\theta$. If $a_{i_0}\theta \rightarrow_{prem(r\theta)} a'_{i_0}\theta$ then immediately $R \models a_{i_0}\theta = a'_{i_0}\theta$.
Otherwise $a_{i_0}\theta \equiv a_{i_0}\theta[c'\tau]$, $a'_{i_0}\theta \equiv a_{i_0}\theta[d'\tau]$ where $C_s \equiv \wedge_i \ a'_i = b'_i \Rightarrow c' = d'$, hence:

1. for all i there exists e'_i, e''_i such that $a'_i\tau \mapsto^*_{R \cup H_j \cup E_j;r\theta} e'_i$, $b'_i\tau \mapsto^*_{R \cup H_j \cup E_j;r\theta} e''_i$ and $e'_i =_{prem(r\theta)} e''_i$.
2. $a_{i_0}\theta \succ \{a'_1\tau,\ b'_1\tau, \ldots, a'_n\tau, b_n\tau\}$.

Let us show that for all i $R \models a'_i\tau = b'_i\tau$: We first prove that $a'_i\tau \mapsto^*_{R \cup H_j \cup E_j;r\theta} e'_i$ implies $R \models a'_i\tau = e'_i$ by induction on $a'_i\tau$ w.r.t. $\succ$. Let $c[g\lambda] \mapsto c[h\lambda]$ be one of the proof step of this derivation.
If $c[g\lambda] \rightarrow_{prem(r\theta)} c[h\lambda]$, we have $R \models c[g\lambda] = c[h\lambda]$.
Otherwise, suppose there is $Q \equiv \wedge_i \ g_i = h_i \Rightarrow g = h \quad \in R \cup H_j \cup E_j$ such that:

1. $g\lambda \succ h\lambda$.
2. $\forall i \ \exists f'_i,\ f''_i$ such that $g_i\lambda \mapsto^*_{R \cup H_j \cup E_j;r\theta} f'_i$, $h_i\lambda \mapsto^*_{R \cup H_j \cup E_j;r\theta} f''_i$ and $f'_i =_{prem(r\theta)} f''_i$.
3. $c[g\lambda] \succ \{g_1\lambda,\ ,h_1\lambda \ldots, g_n\lambda,\ h_n\lambda\}$.

We have: $g_i\lambda \prec c[g\lambda] \preceq a'_i\tau$ and by induction hypothesis we conclude that $R \models g_i\lambda = f'_i$. In the same spirit we have $h_i\lambda \prec a'_i\tau$ and $R \models h_i\lambda = f''_i$. We also have $R \models f'_i = f''_i$. Everything put together we get for all i $R \models g_i\lambda = h_i\lambda$. As a consequence: $R \models c[g\lambda] = c[h\lambda]$ since $R \models_{ind} Q$, and therefore $R \models a'_i\tau = e_i$. In the same manner we can check that $R \models b'_i\tau = e''_i$ and $R \models e'_i = e''_i$. From these remarks we derive for all i $R \models a'_i\tau = b'_i\tau$. Hence $R \models c'\tau = d'\tau$ since $R \models_{ind} C_s$. This implies successively that $R \models a_{i_0}\theta = a'_{i_0}\theta$, $R \not\models a'_{i_0}\theta = b_{i_0}\theta$, $R \models_{ind} C'\theta$.

2.2 If the *case-rewriting* rule applies to C, then we have: $C \equiv \neg(a_k[s_1\lambda_1]_{u_1} = b_k) \vee r \equiv \ldots \equiv \neg(a_k[s_l\lambda_l]_{u_l} = b_k) \vee r$ with R' the following set of rules $\{P_1 \Rightarrow s_1 \rightarrow t_1,\ \ldots,\ P_l \Rightarrow s_l \rightarrow t_l\} \subseteq R$ et $\forall i$ $s_i\lambda_i$ is irreducible by R and does not contain induction variables. The derived clauses are: $\{C_1 \equiv \neg(a_k[t_1\lambda_1]_{u_1} = b_k) \vee \neg P_1\lambda_1 \vee r, \ldots, C_l \equiv \neg(a_k[t_l\lambda_l]_{u_l} = b_k) \vee \neg P_l\lambda_l \vee r\} \cup CNF(P_1\lambda_1 \vee \ldots \vee P_l\lambda_l)$ with $P_i\lambda_i \ll \{s_i\lambda_i\}$ for all i. Let us show that $R \models_{ind} P_1\lambda_1 \vee \ldots \vee P_l\lambda_l$. Assume that there exists a ground R-irreducible substitution τ such that $R \not\models P_1\lambda_1\tau \vee \ldots \vee P_l\lambda_l\tau$ and consider a term $t \equiv s_{i_0}\lambda_{i_0}$ such that no proper subterm of t matches the left-hand side of a rule (just take for t a subterm a_k occurring at a maximal occurrence u_i w.r.t. length). The term $t\tau$ is irreducible at the root since $R \not\models P_1\lambda_1\tau \vee \ldots \vee P_l\lambda_l\tau$. Assume otherwise that there exists a rule $r \in R - R'$ with left-hand side g that applies to $t\tau$ and $t\tau = g\sigma$. Note that every non variable position of g is a non variable position of t since t does not contain induction variables. In particular t is not an instance of g. Since g is linear we can define a substitution by $\rho(x) = t/w$ for every variable x that occurs at some position w of g. We have then $t = g\rho$, in contradiction with the assumption that R' contains all the rules whose lhs matches t .

If the strict subterms of t do not contain defined operators then $t\tau$ cannot be reduced to a non root position since the constructors are free. This is a contradiction with the fact that all defined symbols are completely defined.

If t contains a strict subterm $s = f(t_1, \ldots, t_n)$ with a defined operator f and for all i, $t_i \in T(C, X)$, then s is strongly irreducible (otherwise $s\tau$ is reducible) and does not contain induction variables (by definition of case rewriting). Hence by proposition 4.1. b) there exists a ground instance $t\phi$ of t that is strongly irreducible by R. These remarks implies a contradiction since $t\phi$ contains a defined operator and all defined operators are completely defined. As a consequence, $R \models_{ind} P_1\lambda_1 \vee \ldots \vee P_l\lambda_l$.

Assume that there exists i such that: $R \not\models_{ind} C_i$. In other words there is a ground instance $C_i\theta$ (we can assume that $C\theta$ is ground without loss of generality) such that: $R \not\models_{ind} r\theta$, $R \models a_k\theta[t_i\lambda_i\theta]_{u_i} = b_k\theta$ and $R \models_{ind} P_i\lambda_i\theta$, so $R \models s_i\lambda_i\theta = t_i\lambda_i\theta$. Therefore $R \models a_k\theta[s_i\lambda_i\theta] = b_k\theta$. This implies that $R \not\models_{ind} C\theta$, absurd. The claim is proved.

We show now that the application of an inference rule on C generates only inductive consequences of R.

generate: There are two cases:

- **A.** *generate* applies to $C \equiv \neg(a_{k_0} = b_{k_0}) \vee r$, Let σ_0 be a $S(R)$-substitution of C. If $C\sigma_0$ is not a tautology then again there are two cases:
  - **a.** If there exists a'_0 such that $a_{k_0}\sigma_0 \mapsto_{R[H_j \cup E_j];r\sigma_0} a'_0$, then by claim 2.1, $R \models_{ind} \neg(a'_0 = b_{k_0}\sigma_0) \vee r\sigma_0$.
  - **b.** if *case_rewriting* applies to $C\sigma_0$ then by claim 2.2. the resulting clauses are inductive consequences of R.
- **B.** *case_expand* applies to $C \equiv c_{k'_0} = d_{k'_0} \vee r'$, This case is similar to A.

case simplify: If *case simplify* applies to C the from claim 2.2, only inductive consequences of R are generated.

simplify: If *simplify* applies to C then there are two possibilities:

- **A.** *Simplify* applies to $C \equiv \neg(a_{k_0} = b_{k_0}) \vee r$. In this case we have $a_{k_0} \leadsto_{C_s[R \cup H_j \cup E_j];r} a'_0$ with $R \models_{ind} C_s$. From claim 2.1 we deduce $R \models_{ind} \neg(a'_0 = b_{k_0}) \vee r$.
- **B.** *Simplify* applies to $C \equiv c_{k'_0} = d_{k'_0} \vee r'$. Similar to **A.**

right simplify of constructors: If *right simplify of constructors* applies to $C \equiv f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \vee r$. then consider is a ground substitution σ. Since $R \models_{ind} C\sigma$ we have either $R \models_{ind} r\sigma$ or $R \models f(s_1, \ldots, s_n)\sigma = f(t_1, \ldots, t_n)\sigma$. The first possibility gives immediately the desired conclusion. Hence let us assume the second one. Since f is a free constructor then $\forall i \;\; R \models s_i\sigma = t_i\sigma$. Therefore every clause $Q_i \equiv s_i = t_i \vee r$ verifies $R \models_{ind} Q_i\sigma$.

left simplify of constructors: If *left simplify of constructors* applies to $C \equiv \neg(f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)) \vee r$. Let σ be a ground substitution and let us assume that $R \not\models f(s_1, \ldots, s_n)\sigma = f(t_1, \ldots, t_n)\sigma$. Since $R \models_{ind} C\sigma$ we have either $R \models_{ind} r\sigma$ or $R \not\models f(s_1, \ldots, s_n)\sigma = f(t_1, \ldots, t_n)\sigma$. The first possibility gives immediately the desired conclusion. Hence let us assume the second one. Since f is a free constructor there exists i such that $R \not\models s_i\sigma = t_i\sigma$. Therefore $R \models (\vee_i \neg(s_i = t_i) \vee r)\sigma$.

deletion of a trivial clause and subsumption: If C is deleted then $R \models_{ind} E_{j+1}$ since $E_{j+1} \subseteq E_j$ in this case.

complement: Assume that *complement* applied to $C \equiv \neg(a_k\theta = b_k\theta) \;\vee\; r$ gives $C' \equiv a_k\theta = b'\theta \;\vee\; r$. Let us show $R \models_{ind} C'$. By contradiction assume that $C'\tau$ is a ground instance of C' such that $R \not\models_{ind} C'\tau$. We can also assume that $C\tau$ is ground without loss of generality. Then $R \not\models_{ind} r\tau$ and $R \not\models a_k\theta\tau = b'\theta\tau$. But we also have $a_k = b_k \;\vee\; a_k = b' \;\in\; E \cup H$. Therefore $R \models_{ind} a_k = b_k \;\vee\; a_k = b'$ and it follows that: $R \models a_k\theta\tau = b_k\theta\tau$. This imlpies that $R \not\models_{ind} C\tau$ which is absurd.

Unité de Recherche INRIA Lorraine
Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)

Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399

RR-2015