Generalization Discovery for Proofs by Induction in Conditional Theories

KOUNALIS Emmanuel, URSO Pascal
Université de Nice - Sophia Antipolis, Laboratoire I3S
Dpt. Informatique, Parc Valrose
06108 NICE CEDEX 2, France
kounalis@unice.fr, urso@essi.fr

Abstract

Several induction provers have been developed to automate inductive proofs (see for instance: Nqthm, RRL INKA, LP, SPIKE, CLAM-Oyster, ...). However, inductive theorem provers very often fail to terminate. A proof to go through requires either additional lemmas, a generalization, a suitable induction variable to induce upon , or a case split.

The aim of this paper is to present a simple and powerful heuristic that allows to overcome, in many cases, the divergence of induction provers when working with conditional theories. We first provide a new definition of induction variables and then formalize a new transition rule for induction (named CGT-rule). The essential idea behind it is to propose a generalized form of the conclusion just before another induction is attempted and failure begins. This generalized form is based on the induction hypothesis and the current goal.

CGT-rule enables to prove many theorems completely automatically from the functions definitions alone. We illustrate computer applications to the correctness proof of the insertion sorting algorithm and other programs computing on lists and numbers. All of them have never proved before without user-provided generalizations and/or lemmas.

(Content Areas: Automated Reasoning, Theorem Proving)

Introduction

In this section, we first state the problem of induction, then explain our interest in automated induction, and finally discuss the aims of the present paper and related work.

The Problem

We assume familiarity with the basic notions of *equational logic* and *rewrite systems* (see for instance (Dershowitz and Jouannaud 91)). For simplicity of notation, we assume have only one sort; all the results carry over to many-sorted case without difficulty.

Let A be a set *of conditional equations*, i.e., expressions of the form $e_1 \wedge \ldots \wedge e_n \Rightarrow e_{n+1}$, where e_i are equations which are usually written as $t = s$. $e_1,\ldots, e_n$ are the *premises* and e_{n+1} is the *conclusion* of $e_1 \wedge \ldots \wedge e_n \Rightarrow e_{n+1}$. A *clause* is an expression of the form $\neg e_1 \vee \ldots \vee \neg e_n \vee e'_1 \vee \ldots \vee e'_m$, where $e_1,.., e_n, e'_1,.., e'_m$ are equations. We shall sometimes write this expression in the following equivalent way: $e_1 \wedge \ldots \wedge e_n \Rightarrow e'_1 \vee \ldots \vee e'_m$. We identify a conditional equation and its corresponding representation as an Horn clause. A clause ϕ is an *inductive consequence (inductively valid)* of a set A of axioms if it is valid in the initial model. This will be denoted by $A \models_{ind} \phi$. Inductive consequences are related to deductive consequences as follows: $A \models_{ind} \neg e_1 \vee \ldots \vee \neg e_n \vee e'_1 \vee \ldots \vee e'_m$ if and only if for any ground substitution σ, (for all i : $A \models e_i\sigma$) implies (there exists j such that $A \models e_j\sigma$). For a detailed discussion of initial models, see (Padawitz 88). Since ground terms can easily be well-ordered, induction can be used as a natural technique to prove clauses in the initial model. Unfortunately, there is no a simple proof theory which captures the semantic notion of the initial model. To overcome this problem, one uses formula schemata to formulate an induction rule which is used to prove formulas in the initial model. Morever, no (recursively axiomatizable) set of induction schemata is strong enough to constitute a complete proof theory for the initial model.

To establish inductive consequences, classical theorem proving provides either *explicit induction* ((Boyer and Moore 79), (Aubin 79), (Zhang, Kapur, and Krishnamoorthy 88),...), or *implicit induction* (Dershowitz 82), (Jouanaud and Kounalis 86 and 90), (Kounalis and Rusinowitch 90), (Reddy 90), (Bouhoula, Kounalis, and Rusinowitch 95), (Bouhoula and Rusinovitch 95), (Bonsard, Hasker, and Reddy 96). However, the hardest problems in using either approaches is to find an appropriate induction hypothesis and/or the lemmas needed for the proof. Consequently, proving conjectures in the initial model requires "eureka steps" such as selection of a suitable variable to induce upon, additional lemmas, a generalization, or case analysis for the proof to go through. In some cases a combination of these concepts is needed. In this paper we develop and implement a framework for automating, in some cases, these "eureka steps".

Motivation

The need to be able to prove inductive theorems appears in many applications including number theory, program verification, and program synthesis. For example:

Conditional equations may be used to define arithmetic functions and express suitable properties about these functions. Consider, for instance, the theory A of *nonnegative_integers* with addition, multiplication and exponentiation: $A \equiv \{x + 0 = x,\ x + S(y) = S(x + y),\ x * 0 = 0,\ x * S(y) = x + (x*y),\ EXP(x, 0) = S(0),\ EXP(x, S(y)) = x * EXP(x,$

y)}. Clearly, adding two integers, using the above equations, yields a nonnegative integer. However if we consider the property x+y = x+z ⇒ y = z, its validity cannot be established by merely deductive reasoning. This is equivalent to prove that the assertion is valid (true) in the theory of nonnegative integers which is the initial model of the A (like any well-known property of +, *, and EXP).

Aims of the Paper and Related Works

This paper deals with the problem of proofs of clauses in the initial model of a set of conditional axioms. We first provide an analysis of inductive variables, and then formulate an inference rule for automatically generalizing conjectures. The paper is a successor of (Kounalis and Rusinowitch 90 and 95) and (Bouhoula, Kounalis, and Rusinowitch 95), where induction was formulated as a combination of two concepts: test sets and reduction orderings. Although this method has proved several interesting theorems without assistance, its attempts to prove many theorems fail without additional lemmas and /or suitable generalizations.

Significant effort has been devoted to building heuristics for speculating additional lemmas and for discovering generalizations of *equations in unconditional theories*. The *rippling* heuristic (Bundy et al. 93), the *difference matching* procedure (Basin and Walsh 92), the *divergence critic* (Walsh 94), the use of *proof planning* (Ireland and Bundy 96 and 96a), and the *speculate heuristic* (Kapur and Subramaniam 96). Conceptually, all these approaches relies upon the analysis of failed proof attempts to speculate generalizations and lemmas to complete the proof. All these approaches deals with proofs of unconditional equations in unconditional theories.

The present paper formalizes a new heuristic method that finds out automatically suitable generalizations to prevent proof failure. *The essential idea behind the method is to propose a generalized form of the conclusion just before another induction is attempted and failure begins.* The implication of such a idea is straightforward: it does not need to know that a proof fails, nor the nature of the failure, nor the type of induction being performed. For that purpose a simple and powerful rule is introduced: the CGT-rule (see definition 4). Roughly speaking, the CGT-rule identifies positions in a goal and generalizes the subterms at these positions after filtering them through a set of representative ground terms of TS(A). Further, one of the major problem of inductive theorem provers is to find out the right variable to induce upon. Before going on to the generalization problem, we comment on the use of induction variables by the generate rule, and then carefully define them (see definition 3). The proposed method has been implemented in C. The initial experimentation is very pleasing. For instance, the method can be employed to automatically establish proofs for theorems (see examples) for which all heuristically approaches to automated induction fail to produce a proof.

Outline of our Approach : an Example

To illustrate the essential ideas behind our method let us outline it on a rather complicated conjecture. Suppose our goal is to prove the reflexivity property of the set inclusion relation,

(*) D(L) = true ⇒ SUB(L, L) =true

This is part of a simple *program verification* problem. Consider the following axiomatization A of subset (SUB), element inclusion (∈), all_different_elements (D), wherein a set is represented as a list L :

∈(x,L) = true ⇒ SUB(x.l, L) = SUB(l, L)
∈(x,L) = false ⇒ SUB(x.l, L) = false
SUB(∅,L) = true
E(x,c) = true ⇒ ∈(x,c.L) = true
E(x,c) = false ⇒ ∈(x,c.L) = ∈(x,L)
∈(x,∅) = false
∈(x,L) = true ⇒ D(c.L) = false
∈(x,L) = false ⇒ D(c.L) = D(L)
D(∅) = true
EQ(x,x) = true
true = false ⇒
false = true ⇒

Here, EQ is the identity comparison operator for elements. Note that this simple conjecture is problematic for all automated proof systems (e.g. Nqthm, RRL, INKA, LP, SPIKE, CLAM-Oyster, etc...). None of them can prove it when given just the above definitions. Of course, with the addition of some lemmas all provers are able to prove this. Also, this example cannot be treated by any of the heuristics proposed in *heuristic* (Kapur and Subramaniam 96), (Ireland and Bundy 96 and 96a), (Walsh 94), (Basin and Walsh 92), (Bundy et al. 93). Let see how our method allows an automatic proof of it.

To carry out a proof of (*) we first supply a *well-founded ordering* > on terms (i.e., there is no endless descending chain t1 > t2 >,... of terms). For example, we may compare terms by using the lexicographic path order > generated by the precedence SUB $>_p$ D $>_p$ ∈ $>_p$. $>_p$ ∅. Then SUB(L, L) > true holds in the lexicographic path order even though the two terms presumably have the same semantics (see (Dershowitz 87)). Using a monotonic and stable well-founded ordering > on terms we may apply a (conditional) equation for simplification.

We then compute a *Test Set* TS(A) for A, i.e., a finite set of terms which, in essence, is a finite description of the initial model of A (see for instance: (Kounalis and Rusinowitch 90), (Kounalis 92)). For example, the set TS(A) = {x.l, ∅, true,false} constitutes a suitable test set for A. Note that every ground term is equal to a term over SUB, ., ∈, D, ∅, true, and false. The reason for considering test sets is to instantiate their elements to variables in a conjecture in order to create the induction shemes for the proof of a conjecture.

Finally, a set of transitions rules (see subsection 4.1) is applied to the conjectures; The *generate* rule that allows to derive the induction schemes by instantiating an induction variable (see subsection 4.2) with elements of TS(A). The *case*

analysis rule that simplifies a conjecture with conditional rules and adds to the result the contexts where the respective reductions are valid. The *simplify* rule reduces a conjecture C with axioms from A, induction hypotheses and other conjectures (therefore we can simulate *mutual* induction). The premises 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 *delete rule* that removes trivial conjectures. Notice that this set of inference rules is sound (see for a proof (Bouhoula, Kounalis, and Rusinowitch 95)). The *CGT-rule* that abstracts common subterms in a clause at well-defined positions.

There are two fundamental ideas behind our proof strategy. The first idea is to apply the generate rule on an induction variable (see definition 3). The second idea is to apply (if possible) the CGT-rule just before another induction (generate rule) is attempted. In our example the proof starts with the goal:

(*) $\neg D(L) = true \vee SUB(L,L) = true$

To handle clause (*) we can use the generate rule: we first instantiate the induction variable L with elements of the test set, {x.l, ∅, true,false}, and then simplify (using the rules of simplify and/or case analysis) the resulted instances. The first step gives us two distinct clauses,

1. $\neg D(\emptyset) = true \vee SUB(\emptyset, \emptyset) = true$
2. $\neg D(x.l) = true \vee SUB(x.l, x.l) = true$

Notice that the clause, $\in(x, x.l) = true \vee \in(x, x.l) = false$, is inductive valid in A. So the generate rule gives,

3. $\neg true = true \vee true = true$
4. $\in(x, x.l) = true \Rightarrow (\neg D(x.l) = true \vee SUB(l, x.l) = true)$
5. $\in(x, x.l) = false \Rightarrow (\neg D(x.l) = true \vee false = true)$

To make clauses more readable when the case analysis rule applies we use the notation $p \Rightarrow q$ where p is the rule premise and q the initial clause to which one simplification step was performed.

Conjecture 3 is a tautology, so is deleted. For conjectures 4 and 5 the case analysis rule gives,

41. $EQ(x, x) = true \Rightarrow (\neg true = true \vee \neg D(x.l) = true \vee SUB(l,x.l) = true)$
42. $EQ(x, x) = false \Rightarrow (\neg\in(x,l) = true \vee \neg D(x.l) = true \vee SUB(l,x.l) = true)$
51. $EQ(x, x) = true \Rightarrow (\neg true = false \vee \neg D(x.l) = true \vee false = true)$
52. $EQ(x, x) = false \Rightarrow (\neg\in(x, l) = false \vee \neg D(x.l) = true \vee false = true)$

The premise EQ(x,x)=false in clauses 42 and 52 is simplified using A to true=false and the result reduces by A to a tautology by A . Clause 51 is a tautology. Clause 41 can be rewritten using the case analysis rule to

411. $\in(x, l) = true \Rightarrow (\neg false = true \vee SUB(l, x.l) = true)$
412. $\in(x, l) = false \Rightarrow (\neg D(l) = true \vee SUB(l, x.l) = true)$

Now clause 411 is reduced A to a tautology. Application of CGT-rule to 412 gives (see definition 4),

6. $\neg\in(x, W) = false \vee \neg D(W) = true \vee SUB(W, x.l) = SUB(W, l)$

This is the only clause proposed by CGT-rule. Roughly speaking CGT-rule works in two steps: at the first step it takes as argument an induction hypothesis and a derived clause from it (w.r.t. generate and simplification), and produces a new clause. This new clause is generated in such a way that both sides of a positive literal share non-trivial common subterms. Further, at least one of these subterms is at a position which is a suffix of inductive positition of R (see definition 1). At the second step the GGT-rule generalizes these subterms using a fresh variable as well as all occurrences of these subterms in the whole clause.

To handle clause 6 we can use the generate rule. The first step gives us two distinct clauses,

7. $\neg\in(x, \emptyset) = false \vee \neg D(\emptyset) = true \vee SUB(\emptyset, x.l) = SUB(\emptyset, l)$
8. $\neg\in(x, y.Z) = false \vee \neg D(y.Z) = true \vee SUB(y.Z, x.l) = SUB(y.Z, l)$

which are rewritten using A. Simplification of clause 7 gives a tautology. Simplification of clause 8 gives,

81. $EQ(x, y) = true \Rightarrow (\neg true = false \vee \neg D(y.Z) = true \vee SUB(y.Z, x.l) = SUB(y.Z,l))$
82. $EQ(x, y) = false \Rightarrow (\neg\in(x, Z) = false \vee \neg D(y.Z) = true \vee SUB(y.Z, x.l) = SUB(y.Z, l))$

Now clause 81 is reduced by A to a tautology. Using cases analysis, clause 82 gives

821. $\in(y, x.l) = true \Rightarrow (\neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(y.Z) = true \vee SUB(Z, x.l) = SUB(y.Z, l))$
822. $\in(y, x.l) = false \Rightarrow (\neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(y.Z) = true \vee false = SUB(y.Z, l))$

Clauses 821 and 822 can be rewritten (case analysis), using A, to

821. $\neg\in(y, l) = true \vee \neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(y.Z) = true$ v $SUB(Z, x.l) = SUB(Z, l)$
822. $\neg\in(y, l) = false \vee \neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(y.Z) = true \Rightarrow false = false$

Clause 822 is a tautology. Using Case Analysis, we can reduce Clause 821 to

8211. $\in(y, Z) = true \Rightarrow (\neg\in(y, l) = true \vee \neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg false = true \vee SUB(Z, x.l) = SUB(Z, l))$
8212. $\in(y, Z) = false \Rightarrow (\neg\in(y, l) = true \vee \neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(Z) = true \vee SUB(Z, x.l) = SUB(Z, l))$

The first clause is reduced by A to a tautology. Using the induction hypothesis (6) , 8212 is reduced to

$\neg\in(y, Z) = false \vee \neg\in(y, l) = true \vee \neg EQ(x, y) = false \vee \neg\in(x, Z) = false \vee \neg D(Z) = true \vee SUB(Z, l) = SUB(Z, l)$

which is a tautology. This completes the proof of initial conjecture.

Induction in Conditional Theories : the Machinery

In this section we first provide a set of transition rules for induction with an extention of the case analysis rule, then properly define the notion of induction variable, and finally

present ths CGT-rule.The standard automatic method of performing induction in a given theory A is by means of test sets (see (Kounalis and Rusinowitch 90) and (Bouhoula, Kounalis, and Rusinowitch 95)), or cover sets (see (Zhang, Kapur, and Krishnamoorthy 88), (Reddy 90), (Bonsard, Hasker and Reddy 96)). A *test set* (resp. a *cover set*) is, in essence, a finite description of initial model. In general, a test set (Kounalis 92) is more powerfull concept than the a cover set since, in combination with a ground convergent set of axioms, a test set can be used to refute every false conjectures (Bouhoula, Kounalis, and Rusinowitch 95), (Bouhoula and Rusinovitch 95).

Transition Rules for Conditional Theories

The induction procedure is formalized by the following set of transitions rules (Bouhoula, Kounalis, and Rusinowitch 95), (Kounalis and Rusinowitch 95), (Bouhoula and Rusinovitch 95) applied to pairs of the form (E,H), where E is the a set of clauses which are conjectures to be proved and H the set of inductive hypotheses. The initial set of conditional equations A is oriented with a well-founded ordering to a rewrite system R. The inference system for induction contains the transition rules given below. Roughly speaking, the *generate* rule allows to derive the induction schemes by instantiating an induction variable (see next subsection) with elements of TS(R). These induction schemes are then simplified by R either by conditional rewriting or by case analysis. The resulting conjectures are collected to $\cap_\sigma E_\sigma$. The *case simplify* rule simplifies a conjecture with conditional rules, where the disjunction of all conditions is inductively valid (see definition below). The *simplify* rule reduces a clause C with axioms from R, induction hypotheses from H and other conjectures (therefore we can simulate mutual induction). The rules *Subsume* and *Delete* delete redundants clauses.

Case analysis is a very important rule in the setting of inductive proofs, where case splittig arises naturally from an induction hypothesis. Case simplification illustrates the case reasoning: it simplifies a conjecture with conditional rules provided that the disjunction of the preconditions is inductively valid in R. Let R be a rewrite system for the set of axioms A and let $l \vee r$ be a clause. Define Case_Analysis($l[g\sigma]_q \vee r$) as the set $\{ P_1\sigma \Rightarrow l[d_1\sigma]_q \vee r; \ldots; P_n\sigma \Rightarrow l[d_n\sigma]_q \vee r\}$ if $P_i \Rightarrow g \rightarrow d_i$ is in R and $P_1\sigma \vee \ldots \vee P_n\sigma$ is inductively valid in R.

Example: Let R ≡ { perm?(∅, ∅) = true; perm?(∅, y.L) = false; ∈(x, L) = true ⇒ perm?(x.l, L) = perm?(l, (x, L)); ∈(x, L) = false ⇒ perm?(x.l, L) = false } and let C ≡ perm?(c.Z, ins(c, Y)) = perm?(Z, Y) be a clause. Then Case-Analysis(C) is the set {∈(c,ins(c, Y)) = true ⇒ perm?(Z, del(c, ins(c, Y))) = perm?(Z, Y), ∈(c,ins(c, Y)) = false ⇒ false = perm?(Z, Y) } since ∈(c, ins(c, Y)) = true ∨ ∈(c, ins(c, Y)) = false is inductively valid.

The inference system we use is the following:

- **GENERATE** : (E ∪ {C}, H) $|\!-_\iota$ (E ∪ ($\cup_\sigma E_\sigma$), H ∪ {C}),

  if $C \equiv (a=b)^{\varepsilon} \vee r$ and for all test substitution σ:

  either Cσ is a tautology and $E_\sigma = \emptyset$,

  or $a \rightarrow_{R[HUE];r\sigma} a'$ and $E_\sigma = \{(a'=b\sigma)^{\varepsilon} \vee r\sigma\}$

  otherwise E_σ= **Case Analysis(**Cσ**)**.
- **CASE_SIMPLIFY**: (E ∪ {C}, H) $|\!-_\iota$ (E ∪ E', H)

  if E' = **Case analysis(C)**
- **SIMPLIFY**: (E ∪ {(a=b)$^{\varepsilon}$ ∨ r }, H) $|\!-_\iota$ (E ∪ {(a'=b)$^{\varepsilon}$ ∨ r }, H)

  if $a \rightarrow_{R[HUE];r} a'$, or $a \rightarrow_{HUE[R];r} a' \equiv a[t]_p$ and $p \neq \varepsilon$.
- **COMPLEMENT**: (E ∪ {¬(aσ=bσ) ∨ r }, H) $|\!-_\iota$ (E ∪ {(aσ=b'σ) ∨ r }, H)

  if {¬(b = b') is in R, a=b ∨ a = b' is in E ∩ H and bσ≥b'σ.
- **DELETE** : (E ∪ {C}, H) $|\!-_\iota$ (E, H)

  if C is a tautology .
- **Fail :** (E ∪ {C}, H) $|\!-_\iota$ //.

  if no condition of the previous rules hold for C.

A succesfull defivation is a sequence of transitions (E, H) $|\!-$ (E1, H1) $|\!-$ .... $|\!-$(En, Hn) such that En = ∅. The fail rule is a kind of conjecture disprover. It is very important, when a convergent set of axioms exist for ground terms, to guard against false lemmas. We othen work with *boolean axiomatizations*, that is conditional rules with boolean preconditions over free constructors: Every rule in R is of the type $a_1=b_1 \wedge \ldots \wedge a_n=b_n \Rightarrow s \rightarrow t$ where for all i in [1..n], $b_i \in$ {true, false}. Conjectures can also be *boolean clauses* i.e. clauses whose negative literals are of the type ¬a=b where b∈{true, false}. If we assume that any defined function is completely defined, for any function f with boolean values, the following is inductively valid: $f(t_1,\ldots,t_n)$ = true ∨ $f(t_1,\ldots,t_n)$ = false. We can reformulate the complement rule as follows:

- **COMPLEMENT**: (E ∪ {¬(a=b) ∨ r }, H) $|\!-_\iota$ (E ∪ {(a=$\bar{b}$ ∨ r }, H)

  if b ∈ {true, false}.

Designing Good Induction Schemes

The generate rule captures the essence of induction. It allows to obtain the induction schemes that entails the proof of the conjecture under consideration. An induction scheme must be formed in the way that has a good chance of success. In some sense, the success of a proof is strongly depends to what variable the generate rule applies. Then a question naturally arises: *What is the best variable to replace in order to be able to apply a definition and eventually the induction hypothesis?* For example consider the following set R of conditional axioms that defines the intersection of two lists,

∈(x, L) = true ⇒ INT(x.l, L) → x.INT(l, L)
∈(x, L) = false ⇒ INT(x.l, L) → INT(l, L)
INT(∅, L) → ∅
E(x, c) = true ⇒ ∈(x, c.L) → true
E(x, c) = false ⇒ ∈(x, c.L) → ∈(x, L)
∈(x, ∅) → false
E(x, x) = true
true = false ⇒

Assume we want to prove the following conjectures:

C1 ≡ ∈(x L) = true ∧ ∈(x, S) = true ⇒ ∈(x, INT(L, S)) = true

C2 ≡ INT(X, INT(Y, Z)) = INT(INT(X, Y), Z)

(Boyer and Moore 79) put in evidence the fact that only recursive variables are suitable canditates for variables to induce upon. However, a more profound analysis of inductive proofs, entails that among these recursive variables certain variables are of particular importance. For instance: take C1 and suppose we want to learn whether all ground instances of the conclusion can be proved equal whenever the corresponding ground instances of the preconditions are proved equal. To learn it, we must learn about the values of the variables L and S since variable x is not an induction variable. S is a good induction variable since the definition of INT will apply in the precondition part of C1. However, the definition of INT cannot be used anymore in the conclusion. The reason is that INT is defined by recursion on the first argument. This implies that variable L is *better* to induce upon since it allows the definition of INT to simplify both the precondition and the conclusion parts of C1. Take now C2 and suppose we want to learn whether all its ground instances can be proved equal. To learn it, we must learn about INT(X, INT(Y, Z)) and INT(INT(X, Y), Z). To learn about INT(X, INT(Y, Z)) we must learn about X. But we do not know about X, so X is a canditate induction variable. Similarly, to learn about INT(INT(X, Y), Z) we must know about INT(X,Y). To learn about INT(X, Y) we must learn about X and therefore X is a canditate variable for doing induction. That's why X is the best variable to induce upon.

Definition 1. (inductive position set). If R is a set of axioms and f a function symbol in F, then IP(R,f) ≡{p | p ≠ ε and ∃ (P ⇒ l → r) in R such that l(ε) ≡ f and l(p) ∈ F} is the **inductive position set of function f** w.r.t. R. IP(R) denotes the set {IP(R,f)| f∈ F} and is the **inductive position set** of R.

Example. For the set R of rules that define the intesection of two lists, we get IP(R,INT) ={1}, IP(R,∈) = {2}, IP(R,EQ) = {1,2}.

Definition 1 allows to select among the set of variables of an equation, a subset which is suitable for the application of generate rule. However, we may cut down this set by discarding variables judged useless since they lead to goals to which definitions fail to apply. Let C ≡ $\neg a_1=b_1 \vee \ldots \vee \neg a_n=b_n \vee a_{n+1}=b_{n+1} \vee \ldots \vee a_m=b_m$. We denote by **comp(C)** the set of atoms of C: {t | t is either a_i or b_i for i = 1,2,...,m}.

Definition 2. Suppose R is a left-linear set of conditional rules, and C is a clause. $U_X(t)$ is defined to be the set of positions u in t labelled by x such that whenever there is a rule P ⇒ l → r in R such that l unifies with t/v for some prefix v of u then, |u/v| < D(R) and there is a position p in IP(R,l(ε)) such that p is a prefix of u}. The multiset $U_X(C)$ of variable x in C is then defined as {{ u | u ∈ $U_X(t)$ and t ∈ comp(C) }}.

Example. Suppose C1 ≡ ∈(x, L) = true ∧ ∈(x, S) = true ⇒ ∈(x, INT(L, S)) = true. Then, U_X(∈(x, L)) = ∅, U_L(∈(x, L)) = {2} (with v ≡ ε) U_X(∈(x, S)) = ∅, U_S(∈(x, S)) = {2}, U_X(∈(x, INT(L, S))) = ∅, U_S(∈(x, INT(L, S))) =∅, U_L(∈(x, INT(L, S))) = {21} (here v ≡ 2). Therefore $U_X(C1)$ = {∅}, $U_S(C1)$ = {{2}} and $U_L(C1)$ = {{2, 21}}. Suppose now C2 ≡ INT(X, INT(Y, Z)) = INT(INT(X, Y), Z). Then, U_X(INT(X ,INT(Y, Z))) = {1}, U_Y(INT(X, INT(Y, Z))) = {1}. Notice that the term, INT(X, INT(Y, Z)) unifies with INT(x.l, L), the left-hand side of rule ∈(x,L) = true ⇒ INT(x.l, L) → x.INT(l, L), but Y in INT(X, INT(Y, Z)) is not a suffix of a position in IP(R, INT). U_Z(INT(X, INT(Y, Z))) = {∅}. Similarly, U_X(INT(INT(X, Y), Z)) = {1}, U_Y(INT(INT(X, Y), Z)) = {∅}, and U_Z(INT(INT(X, Y), Z)) = {∅}. Therefore $U_X(C2)$ = {{1, 1}}, $U_Y(C2)$ = {{1}} and $U_Z(C2)$ = {∅}.

The main reason for considering the set $U_X(C)$ is just to select the best variables in a clause to induce upon. The motivation behind the requirements of definition 2 is the following. First, the condition of unifiability is well-understood since if a term t/v does not unify with R, any instance of t/v using the elements of a test set could not be an instance of R. Second, assume that t/v unifies with R: If a variable x is at position u in t/v and the lentgh of u is greater than or equal to the length of the variable positions in R, then further instantiation of x cannot create instances of R since R is assumed to be left linear so eventual substitutions are too shallow to matter. Further, if u is not a suffix of an induction position, then instantiation of this variable cannot gives further information about t/v. The following definition defines formally the notion of induction variables in our setting :

Definition 3 (induction variables): Variable x in C is an **induction variable** if for all variables y in C either x is a generalized variable and y is not a generalized variable or $|U_y(C)| < |U_x(C)|$.

A variable x is said to be generalized if x is the variable introduced by the CGT-rule below, i.e. if x is the variable obtained by replacing common non-trivial subterms in a clause by x. Notice that this requirement is consistent with definition 2 since the CGT-rule generalizes subtems that are at positions suffixes of positions in the induction position set of a function. Notice also that in the case when $|U_y(C)| = |U_x(C)|$ either x or y may be used to induce upon.

Example. Suppose that in C1 no variable is generalized. Since $|U_X(C1)| < |U_S(C1)| < |U_L(C1)|$, L is the induction variable in C1. Assume that in C2 no variable is generalized. Since $|U_Z(C2)| < |U_Y(C2)| < |U_X(C2)|$, X is the induction variable in C2. Suppose now that in C3 ≡ INT(X,W) = INT(W,X) W is generalized and X is not. Then W is the induction variable in C3. However, if we suppose that both X and W are not generalized variables, then either W or X may be considered as induction variable, since $|U_W(C)| = |U_X(C)|$.

Finding Generalizations in Conditional Theories: the CGT-rule

We now introduce an inference rule which turns out to be an indispensable part of practical inductive theorem prover.The essential idea behind the inference rule is to propose a generalized form of the conclusion just before another application of generate rule is attempted and failure begins. At the basis of our method for generalizing is the following

observation: By proving a more general property and less complex (with respect to the number of a function occurrence in a conjecture), we have the advantage of the correspondingly more general induction hypothesis. This would increase the chances of applicability of induction hypothesis to induction conclusion. There are two questions that natural arise:

1. What do we need to know about transforming a clause into a more general one?
2. Which subterm occurences in a clause can we consider as better canditates for generalization?

The CGT-rule below provides a way to transform conjectures into more general ones by abstracting a common non-trivial term that is a suffix of an inductive position. A term is *trivial* if it is a linear variable in a clause. CGT-rule works in two steps: at the first step it transforms a positive literal of a clause B into a literal that both sides share a common non-trivial subterm. At the second step it replaces them with a fresh variable. Notice that these subterms should be both at positions that are suffixes of inductive positions. The reason is that the ground instances of these subterm occurrences "create" in general, the context of the normal forms of the ground instances of the whole term. To create common subterms CGT uses the clause A whose a test set instance of it has been reduced to B. There are two reasons of considering such a clause A . The first reason is that A is "similar" to B, in a sense that some subterms of A may remain invariant during a simplification process; The second reason is that further application of the generate rule to A often creates the same divergence pattern. The following definition captures this discussion.

Let $A \equiv P[x] \vee (l[x] = r[x])$ and $B \equiv Q[s] \vee Q'[s] \vee (a[s]_p = b)$ be two clauses such that x is an induction variable in A. Let s be a maximal (w.r.t. the size complexity) non trivial term in B at position p such that p is a suffix of a position in IP(R), and there is a substitution $\eta\theta$ such that $l\eta\theta \equiv d[s]$ and $P\eta\theta$ subsumes $Q[s] \vee Q'[s]$. Let t be a term in TS(R). Assume that B is derived from $P[x/t] \vee (l[x/t] = r[x/t])$ using the generate and the simplification rules (case simplify or simplify) such that B is not a tautology.

Definition 4 (Generalized Transform): Let A and B be two clauses as above. Assume

1. $b \equiv r\eta$.
2. $(l\eta\theta = r\eta\theta) <_c (l[x/t] = r[x/t])$.
3. $d[s]_q \neq a[s]_p$
4. $Q[g] \vee Q'[g] \vee (a[g]=d[g])$ is inductively valid for any ground term in TS(R).

If $a[W]p \equiv \alpha[(\delta[W])]$ and $l\eta\theta[W] \equiv \alpha[(\gamma[W])]$, for some context $\alpha[]$ with $Var(\alpha[]) \cap Var(\delta[W]) = \emptyset$, $Var(\alpha[]) \cap Var(Q[W]) = \emptyset$, $Var(\alpha[]) \cap Var(Q'[W]) = \emptyset$, and $Var(\alpha[]) \cap Var(\gamma[W]) = \emptyset$, then let D be the clause $Q[W] \vee Q'[W] \vee \delta[W] = \gamma[W]$, otherwise let D be the clause $Q[W] \vee Q'[W] \vee (a[W]p = l\eta\theta[W]q)$. D is said to be a **generalized transform** of B. Variable W is said to be the **generalized variable** in B.

- **CGT-rule :** $(E \cup \{B\}, H) \vdash_t (E \cup \{D\}, H)$ if D is a generalized transform of C.

As we have pointed out above CGT-rules produces a generalized clause in two steps: At the first step the condition 1 with $\eta\theta$ create a clause D' that contains at least an equation (i.e. the equation d[s]=a[s]) in which both sides share the same non-trivial subterm s. Condition 2 implies that a[s] cannot be rewritten to d[s]. Condition 3 implies that both members of equation d[s]=a[s] are distinct. Therefore, the generalized clause cannot be reduced to a tautology. Condition 4 filter out conjectures through a representative set of ground terms to guard against over-generalization. Notice that our motivation is to produce a generalized that is as general as possible. So if D' is a clause of the form $P \vee \alpha[s] = \alpha[t]$, with $Var(\alpha[]) \cap Var(t) = \emptyset$, $Var(\alpha[]) \cap Var(s) = \emptyset$, and $Var(\alpha[]) \cap Var(P) = \emptyset$, $P \vee (\alpha[s] = \alpha[t])$ can be simplied to $P \vee (s = t)$, since $P \vee (\alpha[s] = \alpha[t])$ is inductively valid if $P \vee (s = t)$ is so.

Example : Let $A \equiv perm?(L, is(L)) = true$. So $P[x] \vee (l[x] \equiv perm?(L, is(L))$, $r[x] \equiv true$, and P[x] is empty. Let $B \equiv \neg\in(c, ins(c, is(l))) = true \vee perm?(l, del(c, ins(c, is(l)))) = true$. Then $s \equiv is(l))$, $b \equiv true$ and $Q[s] \equiv \neg\in(c, ins(c, is(l))) = true$. Clearly B is derived from A by using the substitution $\{L/y.l\}$ and rewriting. Then $b \equiv r \equiv true$, and substitution η is the identity. Let θ be the substitution $\{L/l\}$. Then $(perm?(l,is(l))) = true) <_c (perm?(l,del(c,ins(c,is(l)))) = true)$. Let $s \equiv is(l)$. Notice that is(l) is the maximal common subterm. Consider now the clause $D' \equiv \neg\in(c,ins(c,is(l))) = true \vee perm?(l, del(c,ins(c,is(l)))) = perm?(l,is(l)))$. Then the clause $D' \equiv \neg\in(c,ins(c,W)) = true \vee perm?(l, del(c,ins(c,W))) = perm?(l,W)$. Since $\alpha[] \equiv perm?(l,)$ we then have: $Var(perm?(l,)) \cap Var(\vee perm?(l,del(c,ins(c,W)))) = \emptyset$, $Var(perm?(l,)) \cap Var(\neg\in(c,ins(c,W)) = true) = \emptyset$, $Var(perm?(l,)) \cap Var(W)$. Therefore $D' \equiv \neg\in(c,ins(c,W)) = true \vee del(c,ins(c,W))) = W$ is a generalized transform of B and W is a generalized variable.

We list here a representative sample of theorems that can all be proved from the definitions alone, using the CGT-rule. For comparaison, the automated proof systems Nqthm, RRL, INKA, LP, SPIKE, CLAM-Oyster, etc.. failed to produce a proof for all of them when given just the above definitions. Of course, with the addition of some lemmas all provers are able to prove this. All conjectures proposed by the CGT-rule are sufficiently simple to be proved automatically without introducing fresh failure. The definitions of the functions involved into the theorems bellow are classical and therefore ommited.

Example 1: Theorem to be proved: **EQ(y,0) = false** $\wedge$ **div(x,y) = true** $\Rightarrow$ **div(x*k,y)= true.** The test instance substitution $\{k/s(z)\}$ reduces to $EQ(y,0) = false \wedge div(x,y) = true \Rightarrow div(x+x*z,y) = true$. The CGT-rule then speculates the lemma $EQ(y,0) = false \wedge div(x,y) = true \Rightarrow div(x+W,y) = div(W,y)$. This lemma allows the proof of the equation,

EQ(y,0) = false ∧ div(x,y) = true⇒ div(x*k,y) = true to go through without divergence.

Example 2: Theorem to be proved: ∈**(min(L),L) = true**. The test instance substitution {L/x.y.l} reduces to min(y.l) < x = true ⇒ ∈(min(y.l),x.y.l) = true. The CGT-rule then speculates the lemma W < x = true ⇒ ∈(W,x.y.l) = ∈(W,y.l). This lemma allows the proof of the equation, ∈(min(L),L) = true to go through without divergence.

Example 3: Theorem to be proved: **sorted(is(L)) = true.** The test instance substitution {L/y.l} reduces to sorted(ins(y, is(l)) = true The CGT-rule then speculates the lemma sorted(ins(y,W)) = sorted(W). This lemma allows the proof of the equation, sorted(is(L)) = true, to go through without divergence.

Example 4: Theorem to be proved: **perm?(L,is(L)) = true .** The test instance substitution {L/y.l} reduces to ∈(y,ins(y,is(l))) = true ⇒ perm?(L,del(x,ins(x,is(l)))) = true The CGT-rule then speculates the lemma ∈(y,ins(y,W)) = true ⇒ del(x,ins(x,W)) = W.

Example 5: Theorem to be proved: ∈**(x,ins(x,is(l)) = true.** The test instance substitution {L/y.l} reduces to ∈(x,ins(x,ins(y,is(l)))) = true. The CGT-rule then speculates the lemma, ∈(x,ins(x, ins(y, W)) = ∈(x, ins(x, W)). This lemma (with exemple 4) allows the proof of the equation, Perm?(L, is(L)) = true, to go through without divergence.

Examples 3 to 5 form the first completely automatic proof of the correctness of insertion sort.

In the majority of case, the conjectures proposed by CGT-rule are optimal in a sense that are the simplest possible equations to fix divergence and sufficiently simple to be proved automatically without introducing a fresh divergence. When multiple lemmas are proposed, then any one on its own is sufficient to fix divergence. In some cases a false lemma may be also conjectured. The totality of the conditional generalized transforms, during the proof of a conjecture, form a search space. One way of organizing it is by a proof tree: The root node is conjecture C to be proved; every internal node labeled by an equation, has as children all possible generalized transforms of the clause C which is the result of an application of the transition rules to C. This allows to reject quickly all false conjectures.

Conclusion

This paper has described a simple rules which attempts to propose generalizations which may be used to inductive proofs in conditional theories. This support has the advantage of overcoming the usual failures of inductive proofs. The method seems to be very simple and powerful. The applicability of the method is also illustrated with a large number of examples.

References

Aubin, R. 1979. Mechanizing Structural Induction. *Theoretical Computer Science* 9:329-362.

Bronsard F.; Reddy U.; and Hasker R. 1996. Induction Using Term Orders. Journal of Automated Reasoning 16:3-37.

Boyer, R.S., and Moore, J.S. eds. 1979. *A Computational Logic*. NY: Academic Press.

Bouhoula, A., Rusinowitch, M. 1995. Implicit Induction in Conditional Theories. *Journal of Automated Reasoning* 14(2):189-235.

Bouhoula, A., Kounalis, E., and Rusinowitch, M. 1995. Automated Mathematical Induction. *Journal of Logic and Computation* 5(5):631-668.

Bundy A., Stevens A., van Hermelin F., Ireland A., and Smail A. 1993. Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence* 62:185-253.

Basin, D., and Walsh, T. 1992. Difference Unification. *Procedings of Thirteenth International Joint Conference on Artificial Intelligence*.

Dershowitz, N. 1982. Completion and its Applications. *Procedings of the séminaire d'Informatique Théorique*, Paris.

Dershowitz, N. 1987. Termination of Rewriting. *Journal of Symbolic Computation*, 3:69-116.

Dershowitz, N., and Jouannaud, J.P. 1991. Rewriting Systems, *Handbook of Theoretical Computer Science*.

Jouanaud, J.P. and Kounalis, E. 1986. Automatic Proofs by Induction in Theories Without Constructors. *Procedings 1st IEEE Symposium on Logic in Computer Science*. 1990. *Information and Contro*l 82.

Ireland, A., and Bundy, A. 1996. Using Failure to Guide Inductive Proof. *Journal of Automated Reasoning* 16.

Ireland A., and Bundy A. 1996a. Extensions to a Generalization Critic for Inductive Proof. *Proceding of the Thirteenth International Conference on Automated Deduction*.

Kounalis, E. 1992. How to Check for the Ground-reducibility Property in Term Rewriting Systems. *TCS* 106(1):87-117.

Kounalis, E., and Rusinowitch, M. 1990. Mechanizing Inductive Reasoning. *Proceding of the Eigth AAAI* :240-245.

Kounalis, E., and Rusinowitch, M. 1995. Reasoning with Conditional Axioms. *Annals of Mathematics and Artificial Intelligence* 15:125-149.

Kapur, D., and Subramaniam, M. 1996. Lemma Discovery in Automating Induction. *Proceding of the Thirteenth International Conference on Automated Deduction*.

Padawitz, P. 1988. *Computing in Horn Clauses Theories*. Berlin: Springer-Velrag.

Reddy, U. 1990. Term Rewriting Induction. *Procedings of the Tenth International Conference on Automated Deduction*.

Walsh, T. 1994. A Divergence Critic. *Procedings of the Twelfth International Conference on Automated Deduction*.

Zhang, H., Kapur, D., and Krishnamoorthy, M. S. 1988. A Mechanizable induction principle for equational specifications. *Procedings of the Ninth International Conference on Automated Deduction*.