HOW TO FACILITATE THE PROOF OF THEORIES BY USING THe INDUCTION MATCHING. AND BY GENERALIZATION

Jacqueline Castaing »

IRI Bat 490 Univ. de Pans sud F 9140b ORSAY cedex

ABSTRACT

In this paper, we show how we conceive the proof of theorems by sructural induction Our aim is to facilitate the proof of the theorems which can lead, in a context of automatic theorem proving, to very lengthy (or even impossible) proofs
We use a very simple tool, the i-matching or induction-matching, which allows us, on the one hand to define an original procedure of generalization, and on the other hand to define an original way of generating lemmas.

1 Introduction

In Kodratoff-Castaing \A\ , we presented an automatic theorem proving based on the principle of structural induction |3|. We followed on from the works done by Boyer and Moore [2], Aubin I1 | and Huet and Hullot [5]. Our method is now implemented in Lisp (Vax).

Generally, when we want to prove a theorem using structural induction, we first have to prove one or more basic cases and then, one or more induction steps, In our method we try a new and we hope an original approach, to prove the induction steps. We use a very simple tool which we call I matchimg (induction -matching) . Briefly, it is used as follows (for a more detailed explanation see our previous article |4|): let M => N be any induction steps to be proven, M be any hypothesis and N the associated conclusion. If there is any substitution *o* such that : o(M) - N, and the induction variables chosen do not belong to the domain of substi tution, then the induction step M => N is proved. Such a substitution *o* characterizes the induction -matching. It also shows that the hypothesis M is general enough to consider the conclusion N as an instance of M, and to put N among all the accepted hypotheses .

It is obvious that only easy theorems can be proved by induction - matching. If the matching of M toward N fails , we try to remove the causes of the matching failures either by generalizing the theorem, or by generating intermediate lemmas. So, proving theorems in our method is in fact solving the problems of matching.

In this article we intend to show how we understand the problem of the generalization of the theorems. We shall propose two examples With this help, we shall demonstrate that the solution we propose is :

1 clearly justified by our aim of facilitating the proof of theorems

2 clearer than the solutions proposed by Boycr and Moore, Aubin, and also includes them

2 The formal system

We suppose that all the conditions which allow us to manipulate the theorems, and to prove them by using induction in abstract data types are verified . That's to say:

1 the domain (type support) is generated by a family of constructors supposed not to be related We can define, on this domain, the following well-founded ordering, denoted by < ;
x < y iff x is a subterm of y

2 the functions are complicity defined by a set of ax loms [5]. From these axioms, one can obtain a canonical rewrite system named R.

3 the particular properties of the functions given by the users are contained in the set of equations E.

In this paper, we apply our method to the following two examples :

r, • (eql (app x (app x x)) (app(app x x) x))
t_2 : (eqll (rev x) (foo x nil)).

The first theorem shows that the function app (append in Lisp) is associative. The second shows the equivalence between two programs which compute the reverse of a list. The domain is the set of lists of integers , denoted by List. It is generated by the two constructors (nil , cons), where cons is the binary operator cons : Int x List -> List (Int denotes the set of natural integers, generated

♦ New address : LIPN. Univ. de Paris nord
Avenue J-R element F 93400 VILLETANEUSE '

by the two constructors zero and succ).
The proof of theorem t_1 (t_8) is carried out, if we prove:
1 the basic case t_1 (nil) (t_8(nil))
2 the induction step t_1 (x) => t_1 (x <- (cons a x)) (t_8 (x) => t_8 (x <- (cons a x))).

The canonical rewrite system contains the following rules:
r_1 : (eql x x) -> true
r_2 : (eql nil (cons b y)) -> false
r_3 : (eql (cons a x) nil)) -> false
r_4 : (eql (cons a x) (cons b y)) -> (and (eqn a b) (eql x y))
r_5 : (and true y) -> y
r_6 : (and false y) -> true
r_7 : (eqn x x)-> true
r_8 : (eqn zero (succ y))-> false
r_9 : (eqn (succ x) zero)) -> false
r_{10} : (eqn (succ x) (succ y)) -> (eqn x y)
r_{11} : (rev nil)-> nil
r_{12} : (rev (cons a x)) -> (app (rev x) (cons a nil))
r_{13} : (app nil y) -> y
r_{14} : (app (cons a x) y) -> (cons a (app x y))
r_{15} : (foo nil y) -> y
r_{16} : (foo (cons a x) y) -> (foo x (cons a nil))

The set of equations E only contains the equation (app x nil) = x.

When we dispose of a canonical rewrite system, we can evaluate any term M and find its normal form, denoted by Ml, by applying the rules in any order. But, it may happen in some particular case, that we have to define a precise order in the application of the rewrite rules. In this article, we use call- by- need evaluation [7]. We show how we can apply it.
We proceed by steps . First, we try to reduce M at the empty sequence of occurrences, i.e. wc try to apply one of the rewrite rules which gives the definition of the leftmost function symbol in M. If we succeed, we reduce M and obtain a new term M' to which we apply again the same process of evaluation. If we fail, we let (u_1....U_p) be the sequence of the occurrences of subterms of M, which must be evaluated in order to apply one of the previous rewrite rules. Then, we reduce M successively at the occurrences u_1, u_p by applying the same process of evaluation.

The call-by-need evaluation stops either in the case where the normal form of M is an element of the domain, or in the case where the subterms of M to the occurrences u_1...u_p .denoted by M/u_1,....M/U_p, are variables. In this case , to use Aubin's terminology, we say that these variables are recursion variables, or are in the recursion position. Generally, we can also say that the subterms of M which must be evaluated in the call-by-need rule, are in the recursion position.

Example :

We evaluate the term M = (eql (app x (app x x)) (app(app xx)x)).

Step-1 : we try to reduce M at the empty sequence of occurrences. We have to evaluate the subterms of M at the occurrences 1 and 2, in order to apply one of the four rules r_1....r_4, which give the definition of eql.

Step-2 : for evaluating the subterm M/l, we must instanciate the variable at the occurrence 1.1. For evaluating the subterm M/2, we must evaluate the subterm M/2.1,

Stcp-3; we have to instanciatc the variable at the occurrence 2.1.1.

The two occurrences of the variable x, 1.1 and 2.1.1, are in the recursion position.

3 Generalization : existant solutions

The need for an efficient procedure of generalization has been expressed in [1,2,4,8], In such a procedure, we have to define a criterion of choice of the subterms to be generalized in a theorem, verify the new proposition obtained, and give a new way of proceeding if the generalized proposition obtained is a false one. Otherwise, the method proposed fails.

We now analyse Boyer and Moore s and Aubin's solutions.

Boyer and Moore choose the subterms to be generalized according to their syntax. They replace, in the theorem to be proved, some syntatically identical subterms by the same new variables. This simple procedure suits the proving of a large class of theorems, because it is used in a framework of the strategy of cross-fertilization [2]. This strategy reduces the induction steps P' = Q' => P =h (Q') where P'.Q' and P arc recursively defined functions, to a new lemma P = h(P') to be proved. If the term P is defined by g(P'), then the new theorem to be proved g(P') = h(P') can be generalized to g(u) = h(u), where u is a new variable. We must be careful if we want to extend the application of this principle, because the new proposition obtained can be a false one.
For example, if we generalize the theorem t_1 by this principle, the new theorem obtained by replacing the subterms (app x x) by a variable u is in fact a false one.

Thus, we can make some remarks about the Boyer and Moore 's solution. Their principle docs not indicate how to go on if the proposition obtained is a false one. Their principle cannot be fully justified. One can only establish that very often, the generalized theorem can now be proved

Aubin proposes two heuristics of generalization in order to improve on the Boyer and Moore s principle In these two heuristics, he chooses the subterms to be generalized according to their position in the theorem. He distinguishes a recursive function from a tail-recursive function. In practice, the distinction between these two kinds of functions is the presence in a tail-recursive function of accumulators which are variables, whose role is to contain the partial result of the evaluation of the function. For example, the function foo whose definition is given in the paragraph above, is a tail-recursive function, because the variable y is an accumulator, while the function app is a recursively defined one.

For recursive functions, he uses the same criterion for selecting induction variables, and for choosing the subterms to be generalized, He chooses the induction variables among the recursion ones So, he selects among the set of occurrences of the same subterm in the theorem, those which are in position of recursion, and gives them the same name
For example, he generalizes the theorem r, by replacing the occurrences 1.1 and 2 11 of the same variable x , by a new variable u He thus obtains the new proposition to be proved, (eql (app u (app x x)) (app (app u x) x)).

Generally, if the new proposition obtained is a false one, he analyses the subterms which are left in the theorem, and tries to generalize those which are syntatically identical to the recursion subterms.

For tail-recursive functions, Aubin generalizes the constants which are in the position of accumulators. The principle used is called "indirect generalization", because the subterms to be generalized are now in the accumulator position, and not in the recursion position.
Aubin justifies his choice by the following remark : if the theorem to be proved is an equivalence one, the presence of these constants causes the failure of the proof by cross-fertilization.
Let us verify this remark with the help of theorem t_2.

Example:

$_{t2}$ - (eql (rev x) (foo x nil))
Let M = t_2 (x); Let N = t_2 (x <- (cons a x))! = (ecjl (app (rev x) (cons a nil)) (foo x (cons a nil))).
The induction step is M -> N. To cross-fertilize is to apply the hypothesis as a rewrite rule. We can extract two rewrite rules from M. The first one, (foo x nil) -> (rev x), can not be applied, the presence of the constant nil in the accumulator position of the function foo, leads the matching of (foo x nil) with (foo x (cons a nil)) to fail The second rule, (rev x) -> (foo x nil) reduces the induction step to a new lemma, t_a - (eql (app (foo x nil) (cons a nil)) (foo x (cons a nil))).

In order to prove t_3. we have to prove the induction step t_3 ($_x$) => r_3 (x <- (cons b x)). By applying again the same strategy of cross-fertilization, we can now establish that the presence of the constants nil and (cons a nil) at the occurrences 1.1.2 and 2.2 prevents us from using the hypothesis as a rewrite rule. So, the proof of t_2 fails according to the argument given by Aubin.

If the new proposition obtained by replacing the same constants by a new variable is a false one, (and that is the case in our example, the proposition obtained (ecjl (rev x) (foo x v)) being a false one), Aubin proposes to make appear some new constants in the theorem, that he also generalizes. We describe how he proceeds.
He matches M with N and looks for the other causes of the failure Let us suppose that the matching fails because the substitution is attempted on a function symbol. He tries to remove this failure by applying a procedure called a procedure of expansion [6]. He introduces in M the function symbol which is responsible for the failure, and completes its definition in order to obtain a new hypothesis M' equivalent to M. This last condition is satisfied if the function introduced has a neutral element.

Example :

M = (eql (rev x) (foo x nil)).
N = (eql (app (rev x) (cons a nil)) (foo x (cons a nil))).
The matching of M with N fails at the occurrence 1, because the substitution is attempted on the function symbols rev and app, and fails at the occurrence 2.2, for the reason given by Aubin.
According to the procedure of expansion, Aubin introduces at the occurrence 1 the symbol app, and completes its definition by adding the term (rev x) followed by the neutral element of app which is nil.
The new hypothesis obtained is M' - (eql (app (rev x) nil) (foo x nil)). Now, the generalized expression of M', (eql (app (rev x) v) (foo x v)). is the new theorem to be proved.

The two solutions given by Aubin allow us to extend the class of theorems which can be proved. But, we can point out some weaknesses.
The heuristic used for recursive functions is not fully justified. Indeed, the new variable introduced in the theorem is considered as an induction variable at the next step, when the same method is applied again : we do not know whether the proof of the new theorem obtained can now be facilitated or even carried out. Moreover, we do not like the combinatorial aspect which is associated to the choice of the subterms to be generalized.
The second heuristic proposed is more interesting. It is justified by the intention of its autor to use the hypothesis as a rewrite rule. On the other hand, we think that the Aubin's way of continuing, when the generalized proposition is a false one, can be extended without considering the different kinds of functions. The real purpose is to remove the causes of the failure of the matching of M with

N. We now show, using theorem t_1 as an example, that the subterms choosen by Aubin, at the occurrences 1.1 and 2.1.1, are in fact a part of all the subterms which arc responsible for the matching failure.

Example :

M = t_1 (x) = (eql (app x (app x x)) (app (app x x) x)).
N = t_1 (x <- (cons a x))' = (eql (app x (cons a (app x (cons a x)))) (app (app x (cons a x)) (cons a x))) by applying the rules r_4, r_7, r_5, r_{14}.
The matching of M with N fails for the following reasons :

1 we detect a contradictory substitution on the variable x, x <- x and x <- (cons a x) at the occurrences 1.1, 2.1.1. 2.1.2, 2.2.

2 the substitution is attempted on the function symbols cons and app at the occurrence 1.2.
The subterms responsible for the matching failure are at the occurrences 1.1, 1.2, 2.1.1, 2.1.2, 2.2. Aubin generalizes the subterms at the occurrences 1.1, 2.1.1, which are only two causes of the matching failure.

Our method systematizes this approach.

4 Our solution

It consists of two steps, the first one will be exemplified by the proof of theorem t_1 the second one by the proof of theorem t_2.
We assume that the basic cases arc proved
Let M => N be any induction step. If M i-matches with N then the induction step M => N is proved. Let us suppose that the i-matching fails. As we have already indicated in the introduction, the i-matching is a particular matching such that the induction variables do not belong to the domain of of the substitution. So, we can deduce that either the matching fails, or one of the induction variable belongs to the domain of the substitution.
We put in the two lists LM and LN, all the subterms of M and N which are responsible for this failure, each of them being labelled by its occurrence in M or in N.

Let LM = [(M/u_1, u_1),..., (M/u_q, u_q)] and LN = [(N/u_1 u_1),..., (N/u_q, u_q)] be the lists of these subterms.

A. first step : generalization

Deflnition-1 : We say that wc "savagely" generalize the term M at the occurrences $u_1,....u_q$, if we replace the subterms of M at these occurrences by new distinct variables $v_1...,v_q$.

Deflnition-2 : Let M/u_i be an element of the list LM. Let M/ u_1. . . .M/ u_{in} be the set of all the subterms of M which are syntatically identical to the subterm M/ u_i If we generalize M to the occurrences $u_1....u_{in}$, we call the new variables introduced, v_1___v_n, separated variables

Deflnition-3 : We collect a set of separated variables $Vi.....v_{kl}$ if we give them the same variable name, let be u, The susbtitution r - (v, <- u.....v_k <- u) symbolizes this collection.

Basic idea

Broadly speaking, we apply a strategy which generalizes too much, and then find for particular values of the variables, the conditions which make the generalization true So, we savagely generalize M at the occurrences $U_1,....u_q$, given in the list LM. Let MG (X, v_1 ...v_g) be the generalized expression obtained, where X is the set of all the variables of M different from the V_i. Generally, this expression is a false proposition. We look for the conditions on the variables v_i's, so that MG $(X,v_1...,v_g)$ can now be specialized in a new proposition which is true, and upon which we apply our method of proof once again.

In practice, it is very difficult to find the conditions on the V_i. If M contains a predicate of equality, to specialize MG (X, v_1,.... v_q) is in fact to solve some equations of diophantinc type. So, we limit ourselves to simply finding the equality relations between the variables v_t's. We now show how we proceed.

Practical application

We consider the first cause of failure in LM, m = M/u_1.

1 We savagely generalize M at the occurrences of all the subterms of M which are syntatically identical to m. Let $v_1....v_n$ be the sequence of separated variables introduced in M. We successively give the particular values e_1_e_m to the variables of X, and we compute the normal forms MG-1 = MG (X <- e_1, v_1....v_n).....MG-m = MG (X <- em< v_1,.....v_n). These particular expressions only contain separated variables

2 Let i = 1. We apply the call-by-need evaluation to the term MG-i. Let VR_i = (v_i....v_k) be all the recursion variables of MG-i.

3 Let VR_i = (-U,.....v_n) - VR_t be the set of all separated variables which arc left (if any are left). We first collect the variables of VR_i by giving them the same name u, then we also collect the variables of VR_t' by giving them the same name v. Let r be the substitution which symbolizes these two collections.

4 If Mg = τ (MG (X, v_1,..., v_n) is a proposition which is evaluated as true for the particular values of its variables, then Mg is a new theorem to be proved Otherwise, we go on specializing MG (X, v_1,..., v_n) further. We proceed as follows :

- let j be the particular value given to the variable u which falsifies Mg. We instanciate all the variables of VR_i to the value j, and we compute the new normal form : MG-1-j = MG (v_{i1} <- j,..., v_{ik} <- j)! which only contains separated variables of VR_i'. We apply once more the call-by-need evaluation to MG-1-j Let VR_{ij} be the set of recursion variables in MG-1-j Let VR_i = VR_i ∪ VR_{ij} be the union of the two sets VR_i and VR_{ij}. We go on to 3.

It is obvious that this procedure of specialization stops either in the case where the new proposition obtained Mg is now verified, or in the case where we have to collect all the separated variables v_1,..., v_n. In this case, we consider the next term MG-i+1 upon which we apply the same process again. If all the subterms MG-1,...,MG-m have been considered, we consider the second cause of failure which is distinct from m and we go on to 1. If there is no cause left, the procedure of generalization fails, and we go on to the next step.

Example :
M = t_1 (x) = (eql (app x (app x x)) (app (app x x) x)).
N = t_1 (x <- (cons a x))! = (eql (app x (cons a (app x (cons a x)))) (app (app x (cons a x)) (cons a x))).
LM = [(x, 1.1), (x, 2 1.1), (x, 2 1.2), (x, 2.2), ((app x x), 1.2)]
m = x

1 We savagely generalize M at the occurrences of all the subterms which are syntatically identical to x. We obtain MG (v_1, ..., v_6) = (eql (app v_1 (app v_2 v_3)) (app (app v_4 v_5) v_6)). This expression only contains separated variables.

2 We apply the call-by-need evaluation to MG (v_1,...,v_6). The set of recursion variables VR_i contains the variables v_1 and v_4. So, we collect v_1 and v_4 : let u = v_1 = v_4. We collect all the separated variables which are left : let v = v_2 = v_3 = v_5 = v_6. Let τ = (v_1 <- u, v_4 <- u, v_2 <- v, v_3 <- v, v_5 <- v, v_6 <- v) be the substitution which symbolizes these two collections.

3 Let Mg = τ (MG (v_1,...,v_6)) = (eql (app u (app v v)) (app (app u v) v)). Mg is a proposition evaluated as true for particular values of its variables, so, Mg is a new theorem to be proved.
Let us show how we can prove Mg by using the i-matching. The basic case Mg (u <- nil)! is reduced to true by our rewrite system. The induction step Mg (u) => Mg (u <- (cons a u))' is simplified, by our rewrite system, to the form : (eql (app u (app v v)) (app (app u v) v)) => (eql (app u (app v v)) (app (app u v) v)). The hypothesis i-matches the conclusion, so, the induction step is proved, and we have facilitated the proof of theorem t_1.

B. Second step : generating lemmas

We only consider the subterms of M and N which are put in lists LM and LN, and which are not variables. The presence of these subterms in lists LM and LN shows that the substitution has been attempted on the function symbols. We propose to remove them from these lists cither by using the lemmas given in the set of equations E or by using the hypothesis.

Using lemmas given in E

Let M/u_1 and N/u_i be two subterms of M and N put in lists LM and LN such that the function symbols f_i = M (u_i) and g_i - N (tq) at the occurrences u_i of M and N are distinct If we find in E an equation of the form x - (g_i x e), we reduce M at the occurrence u_i using the rule x <- (g_x x c). So, we make appear the function symbol & in M. We proceed in the same way as Aubin, when he applies the procedure of expansion. Let M' be the new hypothesis obtained by reducing M to the different occurrences given in LM and LN. The new induction step is now M' ~> N. If M' 1-matches with N, the proof of the induction step is completed, and so is the proof of M -> N. Otherwise, we put in the two new lists LM' and LN all the causes of this failure, and we come back to the first step, after removing from E all the equations of the form x = (g_i x c) which have been used
If the equations in E do not allow us to reduce M, we go on now, using the induction hypothesis.

Using the induction hypothesis

General case :
Let M = (P m,.....m_p) and N = (P n_1. , n_p), where P is a predicate to be proved Let us suppose that one subterm m_i i-matches with the subterm n_t, with the substitution . It is obvious that M i-matches with the term N' = (P o (m_1)... o (m_p)). So, if we are able to prove the p-1 lemmas o (rrtj) - tij, we will have proved the induction step M => N.

Particular case .
Let M = (P m, m_z) and N = P {n_1 n_2), where P is a predicate of equality. Let us suppose that m_1j and n_x arc put in lists LM and LN, and that tn_1 has one occurrence in n_1. We reduce N to a new term N' by applying the rewrite rule to the occurrence of the subterm m_1 in n_1. We are left with a new theorem to be proved upon which we apply the same method of proof again. As the reader can remark, we proceed as Boyer and Moore do when they cross-fertilize.

Example :

M = t_2 (x) = (eql (rev x) (foo x nil)).
N = t_2 (x <- (cons a x))! = (eql (app (rev x) (cons a nil)) (foo x (cons a nil)))

LM= f((rcvx), 1), (nil, 2.2)],
LN = [((app (rev x) (cons a nil)), 1), ((cons a nil), 2.2)].
E = [(app x nil) = x].
As the reader can remark, we fail on generalizing t_2. So, we reduce M at the occurrence 1 by the rule x -> (app x nil). We obtain the new hypothesis M' = (eql (app (rev x) nil) (foo x nil)). The i-matching of M' with N fails, and LM' is [(nil, 1.2), (nil, 2.2)]. We come back to the first step after removing the equation (app x nil) = x from E. So, E is now an empty set.
We savagely generalize the term M' at the occurrences 1.2 and 2.2. We obtain MG (x, v_1 v_z) = (eql (app (rev x) v_1) (foo x v_2)). We give to the variable x the particular value x = nil, and the normal form of MG (x <- nil, v_1 v_2) becomes (eql v_1 v_g). We collect these two variables which are recursion variables : v = v_1 - v_2. The new theorem to be proved is now t_3 = (ccjl (app (rev x) v) (foo x v)),
Let us prove theorem t_3. The basic case t_3 (x <- nil) is reduced to true by our rewrite system. The induction step is M => N, where

M = t_3 (x) = (eql (app (rev x) v) (foo x v))

N = t_3 (x <- (cons a x))> = (eql (app (app (rev x) (cons a nil)) v) (foo x (cons a v))).

The subterm (foo x v) i-matches with the subterm (foo x (cons a v)) with the substitution *o* = (v <- (cons a v)) (x does not belong to the domain of *a). So,* the new theorem to be proved is now t_4 = (eql (app (rev x) (cons a v)) (app (app (rev x) (cons a nil)) v)). Our procedure of generalization proposes as a new theorem t_5 = (eql (app u (cons a v)) (app (app u (cons a nil)) v)), which can now be proved by using the i-matching. We have facilitated the proof of theorem t_2.

Conclusion

We can now specify the reasons which allow us to think that our method includes those of Boyer and Moore and Aubin.

1 Our procedure of generalization contains the two heuristics used by Aubin for the choice of the subterms to be generalized.

2 We use the same strategy of proof as these autors when we apply the procedure of expansion, or when we cross-fertilize.

6 References

[I] Aubin R. : "Mechanizing structural induction ". Ph.D ; thesis . Univ Edinburgh (1976).

[2] Boycr RS and Moore J S. : "A computational logic ". Academic Press (1980).

[3] Burstall R. : "Proving propreties of program by structural induction". Computer J. 12 (1) (1969) p 41 - 48.

[4] Kodratoff Y. and Castaing J. : "Trivializing the proof of trivial theorems ". IJCA1 (1983). Kalsruhe West Germany. Proceedings of the eight international joint conference on Artificial Intelligence p 930.

[5] Huet G. and Hullot J-M. : "Proofs by induction in Equational theories with constructors". 21th IEEE Symposium on Foundations of computer Science (1980).

[6] Wegbreit B. : "Goal directed program transformation ". IEEE Trans. Softw. Eng. SE -2.2 Q 71) p 69-80.

[7] Manna, et al. : "Inductive methods for proving properties of programs ". CACM, vol 16, no 8, (1973).

[8] Abdali, et al. : "Generalization heuristics for theorems Related to recursively Defined Functions ", Proc. 4th National Conf on Artificial Intelligence, Austin, (1984).