

INDUCING FUNCTION PROPERTIES FROM COMPUTATION TRACES

Pierpaolo Degano
Istituto di Scienze della Informazione
Università degli Studi di Pisa
Corso Italia 40
56100 Pisa, Italy

Franco Sirovich
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Via S. Maria 46
56100 Pisa, Italy

A system which induces properties of functions is presented. Induction is performed by step-wise generalizing specific given elements of the function domains for which the system can test that the property holds. The system relies on symbolic computation, reflexivity lemmas, and an estimate of the behaviour of the functions. Finally, the paper gives a basis for an evaluation of the system by constructively defining a class of theorems which the system is able to induce.

"In mathematics as in physical sciences we may use observation and induction to discover general laws. But there is a difference. In the physical sciences, there is no higher authority than observation and induction, but in mathematics there is such an authority: rigorous proof." G. Polya How to solve it. Doubleday (New York, 1957).

1. INTRODUCTION

Inductive reasoning has been a popular task in Artificial Intelligence since the very early beginning. It has been attempted in such domains as (just to mention a few) series completion, grammar inference, automatic programming, and theory formation [5,6,9,10,14,17,19,20,22].

In deductive sciences, inductive reasoning plays a peculiar role in the process of knowledge development. In fact, the correctness of an induction can be confirmed by a rigorous proof, even though only a semi-decision procedure might be available, depending on the formal system complexity. Moreover, the rejection of an inductive hypothesis may bring forth explanations about the failure which help in refining the hypothesis.

It goes far beyond the scope of this paper to give a complete account of all the work on inductive reasoning in formal systems. We might mention just a few references. Meltzer [15], Michalski [16], Plotkin [18] and Vere [21] have tackled various forms of inductive reasoning tasks in the predicate calculus. Brown and Tarnlund [4] were concerned with finding a close form solution to difference equations. In the

context of program verification, Boyer and Moore [2], Brotz [3], and Aubin [1] generalize theorems which are then proven by a suitable induction principle. The problem domain we have chosen is akin to the latter.

The system we present induces properties of recursive functions. The properties are defined to the system as (boolean valued) recursive functions. The system can observe that a given property holds for given elements of the domains of the involved functions (called ground property), and examine the corresponding ground property computation. Since the computation is indeed a proof, the evidence the system starts from is that a very trivial theorem holds. The definitions of properties and functions are available and must be taken into account, because no relevant inductive reasoning can be made irrespectively of their nature. Secondly, induction can depend on the details about the flow of computation. We simply provide the system with the kind of information one might get from any tracing facility, i.e. which function, when and where, was applied during the computation.

Thus, inductive reasoning requires the system to grasp a notion of function definition and computation up to be able to formulate a theorem about functions at an expertise level. We

do not claim that a competent system which induces function properties does understand functions and computability. The system we present here is not at all able to develop a theory of computation, hence it does not compare with systems for scientific discovery, such as Lenat's AM [12]. Nonetheless, it exhibits some competence in function property induction.

Again, let us stress the advantage we gain from addressing an inductive reasoning task in the framework of a formal system. The aptness of the inductive system can be precisely stated, confirmed by a formal proof, and anyway appreciated on firm grounds (as far as the observer understands the formal function and property definitions). In this prospect, we cared for providing a precise evaluation of our inductive method effectiveness, and we proved that the method is strong enough to induce formulae which are actually theorems if the involved functions and properties belong to a constructively defined class.

2. A FORMAL CALCULUS ENVIRONMENT

We use a simple recursive function formalism, TEL (Term Equation Language) which was introduced [13] for proving program properties by symbolic computation (e.g. [2]) and is similar to other independently developed formalisms [1,7]. For the present application we add types to TEL so that the resulting language is so similar to Aubin's that the formal treatment and all results of his carry over Typed TEL (later referred to as TTEL). We now briefly overview TTEL, borrowing some nomenclature from Aubin's.

Every term, i.e. every variable and function application, has a type. Each type is defined by a set of type equations which also define the type constructors. All the types and constructors occurring in the equation (apart from the type and constructor being defined) must be defined at a lower level. The language is quantifier free, because any variable occurring in an equation is implicitly universally quantified over its type. Examples are

```
(TYPE(TT())=B; TYPE(FF())=B)
(TYPE(O())=N; TYPE(S(N))=N)
(TYPE(NIL())=L; TYPE(C(N,L))=L)
(TYPE(LNIL())=LL; TYPE(LC(L,LL))=LL)
```

The constructor TYPE is used only to denote type equations. In the following we will omit the argument list of 0-adic constructors. A type is called reflexive if it is defined in terms of itself (e.g. N, L and LL). Analogously, constructors like S and C are reflexive and the argument position where the type they construct occurs is called the reflection argument position. Non-reflexive constructors will also be called terminators of the type. Given a term $c(t_1, \dots, t_n)$ where c is a constructor, $t_1, \dots, t_n$ are terms, then t_i ($1 \leq i \leq n$) is an immediate predecessor of $c(t_1, \dots, t_n)$ if t_i occurs in a reflexion argument position.

Defined functions are introduced by stages. A function definition is a pair, whose first component is a type equation which defines the types of the arguments and the type of the result. For example $\text{TYPE}(\text{EQN}(N, N)) = B$. The second component consists in a set of equations (rewrite equations), which allows a definition by cases [7,11]. Rewrite equations obey the schema $f(a_1, \dots, a_n) = \langle \text{rewriting term} \rangle$ where f is the function being defined, and $a_1, \dots, a_n$ are terms (formal arguments) which may consist either in a variable, or in a constructor applied to variables only (recursion arguments). The only variables which may occur in the $\langle \text{rewriting term} \rangle$ are the formal argument variables. If f occurs in the $\langle \text{rewriting term} \rangle$ (recursive equation), its recursion arguments must be immediate predecessors of the formal recursion arguments.

The definition by cases is restricted as follows. If an argument position is recursive in one equation, then it must be a recursive argument position for all the equations, and for each constructor of the required type there must be exactly one rewrite equation. If the function simultaneously recurs on two or more argument positions, then exactly one rewrite equation must be given for each tuple of constructors of the required types (see the example below).

The metalinguistic constraints allow the definition of total functions only. Nevertheless, all primitive recursive functions over the type of natural numbers N can be defined. This is concrete example.

```
{EQN(O,O)=TT;                               (ENb1)
 EQN(O,S(y))=FF;                             (ENb2)
```

EQN(S(x),O)=FF; (ENb3)
 EQN(S(x),S(y))=EQN(x,y)} (ENr)

The definition of computation of a TTEL term follows.

- i) Specialize a rewrite equation so that its left-hand side becomes identical to a (sub)term of the evaluating term;
- ii) Substitute in the evaluating term the specialized equation right-hand side for the (sub)term;
- iii) Repeat from Step (i) until no equation left-hand side can be made identical to a (sub)term of the evaluating term.

The interpreter adopts the call-by-need computation rule.

Let us now extend the TTEL term definition by introducing free typed variables. A free variable can be instantiated to any term of its type, possibly introducing new free variables. A symbolic term is a term in which free variables occur, otherwise the term is ground. We can extend the definition of computation to handle symbolic evaluating terms. Step (i) only needs to be extended so that the involved test for identity can cause evaluating term free variables to be instantiated. The computation of a symbolic term may be non-deterministic, due to the inherent non-determinism of free variable instantiation. Thus, symbolic computations are (possibly infinite) trees. A concrete example is the following. The term EQN(S(x),S(S(y))) (free variables will be underlined) is reduced to EQN(x,S(y)) by (ENr) and then either to FF if x is instantiated to O by (ENb2), or to EQN(x1,y) if x is instantiated to S(x1) by (ENr). From this point on, all EQN equations can be applied.

We might provide the same inference rules available in Aubin's system. We omit them here since we are not interested in the deductive aspects of the formalism. It is only important to notice that ground property computations yielding TT can be seen as proofs of the property because the interpreter itself implements the tactics Aubin calls simplification. A theorem with universally quantified variables is proven by straight computation if when free variables are substituted for quantified variables, the obtained symbolic term computation yields TT without instantiating the introduced free variables.

3. AN INDUCTIVE REASONING TASK IN TTEL

The system accepts TTEL type and function definitions, and can ask the TTEL interpreter to compute a ground property. The interpreter provides the system with the resulting value and with a computation trace consisting of a sequence of pairs (rewritten subterm, applied rewrite equation). The goal is to induce from this evidence a formula which subsumes the ground property, and hopefully is a theorem itself.

First, the system finds the most general theorem whose proof consists in the given computation trace. This is a proper generalization and the system is able to check it. Secondly, the system exploits properties of predicates such as equality to further generalize the theorem. The system does not give a proof of the new theorem, but a proof could easily be obtained by simplification and induction. A final generalization is obtained on the grounds of few rules which might instead yield non valid results.

4. GENERALIZATION BASED ON PROOF

The first clue the system starts from is the given computation trace. In fact, the very same computation trace may prove a theorem stronger than the given one. In the first place, if a function application term is never evaluated during the computation, the term can safely be substituted by a universally quantified variable and the given computation will still be a proof of the obtained theorem. A typical example is provided by the following ground property (function definitions are given in Appendix I)

EQLN(A(C(PLUS(S(O),O),NIL),
 C(S(O),C(O,NIL))),
 A(C(S(O),C(O,NIL)),
 C(PLUS(S(O),O),NIL))). (1)

Since the terms PLUS(S(O),O) are never evaluated, they can be substituted by a universally quantified variable, yielding the theorem
 EQLN(A(C(x,NIL),C(S(O),C(O,NIL))),
 A(C(S(O),C(O,NIL)),C(x,NIL))).

The same generalization can be done on those data terms (i.e. terms built on constructors only) whose structure is irrelevant to the computation. The system can distinguish the relevant part of the occurring data terms by resorting to the function rewrite equations, whose formal argument terms describe exactly the most general data term the rewrite equation can be

applied to. The required generalization is obtained by firstly substituting free variables for the maximal data terms in the theorem. Then the resulting symbolic term is evaluated by forcing the computation to exactly follow the given computation trace. The original free variables will turn out to be instantiated only as far as needed to obtain the given computation. Let us work out the above example to clarify the matter.

Suppose the maximal data terms are replaced as $EQLN(A(\underline{v1}, \underline{v2}), A(\underline{v2}, \underline{v1}))$. (2)

Since the computation trace reports that the recursive A equation was applied to the A terms, the variables $\underline{v1}$ and $\underline{v2}$ need to be instantiated to $C(\underline{v11}, \underline{v12})$ and $C(\underline{v21}, \underline{v22})$ respectively. Analogously, $\underline{v12}$ is subsequently instantiated to NIL and $\underline{v22}$ to $C(\underline{v23}, \text{NIL})$. Collecting all instantiations, and substituting the still remaining free variables by universally quantified variables, we obtain the theorem $EQLN(A(C(\underline{v11}, \text{NIL}), C(\underline{v21}, C(\underline{v23}, \text{NIL})))),$

$A(C(\underline{v21}, C(\underline{v23}, \text{NIL})), C(\underline{v11}, \text{NIL})))$.

However, if the starting term is $EQL(A(\underline{v1}, \underline{v2}), A(\underline{v3}, \underline{v4}))$ the forced computation is not able to reconstruct the identities of $\underline{v1}$ to $\underline{v4}$, and of $\underline{v2}$ to $\underline{v3}$. Consequently, the substitution of free variables for input data terms must be done carefully.

The heuristic rule for free variable introduction adopted by the system is based on the following observation. In order to have interesting theorems, the variables occurring in one actual argument term of functions such as EQLN (called balanced functions) should occur in the other argument term as well. The system is able to recognize balanced functions from their definitions (basically because they recur simultaneously on both argument positions). Moreover, the user is allowed to tag functions as being balanced (e.g. the boolean functions AND and IMPLIES may conveniently be handled as balanced functions). Thus, the system substitutes identical data terms with different free variables in one argument term, and if the involved function is balanced, carries the introduced variables over the other argument term. For example, if the ground property is $EQLN(A(C(\text{O}, \text{NIL}), C(\text{O}, \text{NIL})), A(C(\text{O}, \text{NIL}), C(\text{O}, \text{NIL})))$ then free variables are introduced as follows $EQLN(A(\underline{v1}, \underline{v2}), A(\underline{v2}, \underline{v1}))$

and forced computation yields the theorem $EQLN(A(C(\underline{v11}, \text{NIL}), C(\underline{v21}, \text{NIL})),$
 $A(C(\underline{v21}, \text{NIL}), C(\underline{v11}, \text{NIL})))$.

Actually, $\underline{v1}$ and $\underline{v2}$ in the second A term could also be introduced the other way around, thus yielding a trivial theorem. The system produces all possible free variables introductions. Some of them are proven by a triviality checker, others are discarded by subsumption.

The limitation of the proof based generalization method stems from its strong dependence on the involved functions. For example, consider the following ground property.

$EQL(A(C(\text{O}, \text{NIL}), A(C(\text{O}, \text{NIL}), R(\text{NIL}))),$ (3)
 $A(A(C(\text{O}, \text{NIL}), C(\text{O}, \text{NIL})), R(\text{NIL})))$.

No free variable introduction helps, because the forced computation will anyway yield back the ground property. In fact, EQL is a much more demanding equivalence relation than EQLN, because EQL accurately checks the list elements by means of EQN.

The generalization method presented in the next Section exploits the presence of equivalence predicates in the theorem, and it is a first start on generalizing the proof.

5. GENERALIZATION BASED ON EQUIVALENCE

Let us describe the role of equivalence by means of example (3). The forced computation starts with the following term (called symbolic input term).

$EQL(A(\underline{v1}, A(\underline{v2}, R(\underline{v3}))), A(A(\underline{v1}, \underline{v2}), R(\underline{v3})))$.

The computation trace forces the instantiation of $\underline{v1}$ to let the outermost A terms produce (through the recursive A equation) two C terms.

Thus the symbolic term is rewritten as follows $EQL(C(\underline{v11}, A(\underline{v12}, A(\underline{v2}, R(\underline{v3}))))$
 $C(\underline{v11}, A(A(\underline{v12}, \underline{v2}), R(\underline{v3}))))$.

The EQL recursive equation is applied yielding $AND(EQN(\underline{v11}, \underline{v11}), EQL(A(\underline{v12}, A(\underline{v2}, R(\underline{v3}))),$
 $A(A(\underline{v12}, \underline{v2}), R(\underline{v3}))))$.

This is the first place where the notion of equivalence can help generalizing. The system notices the occurrence of the term $EQN(\underline{v11}, \underline{v11})$ whose corresponding evaluation in the computation trace yields TT. Since EQN is an equivalence relation, the reflexivity lemma $EQN(x, x)$ can easily be proven, and used to rewrite $EQN(\underline{v11}, \underline{v11})$ to TT. The part of the computation trace describing the evaluation of EQN can be by-passed thus leaving the variable $\underline{v11}$ free.

More precisely, whenever an equivalence predicate application $e(t_1, t_2)$ is found during the forced computation, the system attempts to use the reflexivity lemma $e(x, x)$. If t_1 and t_2 are identical, and are sub-terms of the symbolic input term, a new free variable is introduced for both t_1 and t_2 .

In the given example, the computation proceeds with the evaluation of the EQL term, and by instantiation of v_{12} to NIL yields $EQL(A(v_2, R(v_3)), A(v_2, R(v_3)))$.

The reflexivity lemma is not applied because the second A term is not part of the symbolic input term but it comes from the computation of the term $A(v_1, v_2)$. On the contrary, after a few steps the variable v_2 ripples out and the evaluating term becomes $EQL(R(v_3), R(v_3))$.

Both $R(v_3)$ terms satisfy the conditions above, and the system generalize them. Finally, collecting all instantiations, the induced theorem is the following

$$EQL(A(C(v_{11}, NIL), A(C(v_{21}, NIL), v_4)), \quad (4) \\ A(A(C(v_{11}, NIL), C(v_{21}, NIL)), v_4)).$$

The given computation trace is no more a proof of the induced theorem. Yet, a proof could easily be obtained by a simplification inference rule, and by use of reflexivity lemmas. Hence, the induced formula is actually a theorem.

Since the system exploits only the reflexivity property of the equivalence relations, any reflexive predicate can be handled this way. We did not attempt to let the system discover reflexivity by itself from the predicate definitions. However, the system is able to induce such a property.

Although the formula induced at this point is valid, it is a poor generalization, because the structure of the given data terms may still over-specialize the theorem. We would like for example the C terms in (4) to be generalized, thus obtaining the associativity theorem for A.

6. GENERALIZATION BASED ON ESTIMATED FUNCTION BEHAVIOUR

The data structures still present in the theorem mirror (part of) the structure of the given data terms. A further (and possibly bold) generalization is now required. The first problem is to select data terms as candidates for generalization. The second problem is to check that

candidate modifications do not change the computation structure.

The system looks for those data terms which are so general that they appear as "skeletons" for their type. A skeleton of type T is a data term such that: i) All constructors of type T occur in it; ii) In the argument positions of type T $\neq$ T a variable of type T occurs.

Examples of skeletons are $C(v_1, C(v_2, NIL))$ and $S(0)$, while the following data terms do not classify as skeletons: TT , because it does not contain the constructor FF, and $C(S(0), C(v_1, NIL))$, because $S(0)$ is not a variable.

Focussing on skeletons embodies a peculiar notion of computation. First of all, the "structure" of data terms and the "structure" of computations are considered to be strongly interrelated, so that the forced computation is believed to be able to reconstruct the details of data terms which are necessary and sufficient to determine the computation flow. Secondly, the skeleton definition requires all constructors for the type to occur in a skeleton, in the assumption that if a constructor would not occur, the given computation trace would not be a representative of all the possible computation traces. Finally, the hierarchical definition of types is believed to induce a corresponding nesting in the computation, and therefore will require hierarchical generalizations, as we will see later.

Still, two different skeletons may be related to each other in such a way that their generalization to two different variables would not be valid. The system adopts just a naive notion of relationship, i.e. two skeletons are related iff they share a universally quantified variable.

Good candidates for generalization are defined to be the maximal unrelated skeletons. However, the given theorem may hold only if the actual data terms obey exactly the structure described by the corresponding skeletons. A classical example is the following

$EQN(PLUS(S(0), S(S(0))), TIMES(S(0), S(S(S(0)))))$. Hence, the system must test whether candidate skeleton modifications appear to bring about computation structure modifications.

The system test is based on those functions, such as EQN, which simultaneously recur on the-

ir arguments. Simultaneously recurring functions (srf's) implicitly define a measure on the argument values (i.e. the number of recursive applications which are needed to completely "peel down" an argument value). The difference between the argument measures determines which srf base equation is applied to terminate the computation. If the actual argument measures depend at different degree on the measure of the candidate skeleton, a skeleton modification may cause a different base equation to be used to terminate the (modified) computation. To be on the safe side, the system discards the candidate.

The measure of a data term cannot be computed a priori since it is defined by the rewrite equations of the functions which recur on it. In the current implementation a drastic simplification is made, and only data structures which are reflexive on one argument position are tested. Moreover, the actual argument terms of a srf will in general be function application terms (just like in the example). Hence, the system must be able to estimate how functions transform data term measures.

For each function, the system computes an arithmetical expression (called norm) which, roughly speaking, expresses the measure of the function value in terms of the measure of its arguments. Function composition in the argument terms of the srf can be accounted for by introducing two suitable auxiliary functions (defined by composition) such that the srf argument terms can be rewritten as applications of the auxiliary functions to data terms only. The auxiliary function norms tell the system how the measures of the the outermost predicate argument values depend on the measures of the skeletons. Since we want to capture the effect of the modification of a specific skeleton, the partial derivatives of the auxiliary function norms w.r.t. the argument position under generalization are computed. The generalization is accepted iff the derivatives can be simplified to the same expression. Note that such a test brings together the function definitions, the form of the theorem, and the computation structure.

For instance, let us consider the candidate $S(0)$ in the example above. The norm of PLUS is $p_1 + p_2$ (see below), and the norm of TIMES is $p_1 * p_2$ (the variable symbol p_i occurring in norm

expressions stands for the measure of the i -th argument term). The partial derivatives of $p_1 + p_2$ and $p_1 * p_2$ w.r.t. p_1 are 1 and p_2 , thus $S(0)$ is not generalized.

Since constructors may occur in function definitions, norms for constructors must be given (which, by the way, define the measures of data terms). The norm of a constructor is defined to be 0 if the constructor is not reflexive, otherwise $1 + p_i$ if the constructor is reflexive in the argument position i . The norm of a function is computed from the function definition and from the norms of the occurring functions. Basically, the norm of the function used in the base equation is combined with the norm of the function used to perform recursion where the operators are substituted by higher rank operators. The algorithm for computing the norm [8] is rather intricate because many heuristics have been designed to cope with the complexity of function definitions. The computed norm is actually satisfactory for a wide class of functions (examples are reported in Appendix II). However, the estimate of decreasing functions, and of functions with more than one recursive or base equations are far from being adequate.

We can now describe the stepwise generalization method based on estimated function behaviour. Candidate skeletons are recognized in the given theorem, and each candidate is tested as follows. The computation trace is analyzed to pick out those srf application terms in which the candidate occurs. If no such term is found, the candidate skeleton is generalized. If for each srf application term the partial derivatives of the arguments w.r.t. the candidate are the same, then the candidate is generalized, otherwise it is discarded. Candidate generalization may introduce new candidate skeletons which are in turn tested and generalized as above.

7. A COMPLETE EXAMPLE

Let us consider the following ground property

```
EQS(LR(LA(LC(C(PLUS(S(0),0),C(S(0),NIL))), (E1)
      LNIL),
      LC(C(PLUS(S(0),0),C(S(0),NIL))),
      LNIL))),
  LA(LR(LC(C(PLUS(S(0),0),C(S(0),NIL))),
      LNIL)),
  LR(LC(C(PLUS(S(0),0),C(S(0),NIL))),
      LNIL))).
```

The term $\text{PLUS}(S(0),0)$ is never evaluated. A free variable $\underline{nx}$ is substituted for it, and free variables $\underline{nllx}$ and $\underline{nllly}$ are introduced to force computation yielding the symbolic terms

$\text{EQS}(\text{LR}(\text{LA}(\underline{nllx}, \underline{nllly})), \quad (\text{E2})$
 $\text{LA}(\text{LR}(\underline{nllx}), \text{LR}(\underline{nllx})))$ and

$\text{EQS}(\text{LR}(\text{LA}(\underline{nllx}, \underline{nllly})), \quad (\text{E3})$
 $\text{LA}(\text{LR}(\underline{nllx}), \text{LR}(\underline{nllly}))).$

The forced computation of (E2) gives

$\text{EQS}(\text{LR}(\text{LA}(\text{LC}(\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL})), \text{LNIL})), \quad (\text{E4})$
 $\text{LC}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{LNIL}))),$
 $\text{LA}(\text{LR}(\text{LC}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{LNIL})),$
 $\text{LR}(\text{LC}(\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL})), \text{LNIL}))).$

The type LL terms are due to the applications of LA equations, while the type L terms come from the applications of EQLN equations. The variables appear because EQLN checks for L length equality only. The forced computation of (E4) gives

$\text{EQS}(\text{LR}(\text{LA}(\text{LC}(\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL})), \text{LNIL})), \quad (\text{E5})$
 $\text{LC}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{LNIL}))),$
 $\text{LA}(\text{LR}(\text{LC}(\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL})), \text{LNIL})),$
 $\text{LR}(\text{LC}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{LNIL}))).$

The generalization method based on equivalence is applied to (E4). The reflexive predicate EQLN application term

$\text{EQLN}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})))$
 was reduced to TT (note the role of LR). Since the argument terms are identical and do occur in (E4), they are generalized to $\underline{nly}$. Analogously, the data term $\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL}))$ is generalized to $\underline{nlx}$. The following theorem is thus induced
 $\text{EQS}(\text{LR}(\text{LA}(\text{LC}(\underline{nlx}, \text{LNIL}), \text{LC}(\underline{nlx}, \text{LNIL}))), \quad (\text{E6})$
 $\text{LA}(\text{LR}(\text{LC}(\underline{nly}, \text{LNIL}), \text{LR}(\text{LC}(\underline{nlx}, \text{LNIL}))).$

When the method is applied to (E5), the following EQLN term is found

$\text{EQLN}(\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL})), \text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL}))) \quad (\text{E5.1})$
 Since the argument terms are not identical, (E5) is correctly no more generalized.

Finally, the generalization method based on estimated function behaviour is applied to (E6). The skeletons are $\text{LC}(\underline{nlx}, \text{LNIL})$ and $\text{LC}(\underline{nly}, \text{LNIL})$, and EQS is the only srf which applies to terms containing them. Since the partial derivatives of $\text{LR}(\text{LA}(p1, p2))$ and $\text{LA}(\text{LR}(p1), \text{LR}(p2))$ w.r.t. $p1$ are both 1, the skeleton $\text{LC}(\underline{nlx}, \text{LNIL})$ is generalized to $\underline{nllx}$. $\text{LC}(\underline{nly}, \text{LNIL})$ is analogously generalized to $\underline{nllly}$, and the system induces the following theorem

$\text{EQS}(\text{LR}(\text{LA}(\underline{nllx}, \underline{nllly})), \text{LA}(\text{LR}(\underline{nllly}), \text{LR}(\underline{nllx}))).$
 When (E5) is considered, the skeletons $\text{C}(\underline{nx1},$

$\text{C}(\underline{nx2}, \text{NIL}))$ and $\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL}))$ are not generalized. In fact, the srf EQLN is directly applied to them (see (E5.1)), i.e. we have a term of the form $\text{EQLN}(p1, p2)$. The partial derivatives of $p1$ and $p2$ w.r.t. $p1$ are 1 and 0, thus the skeleton $\text{C}(\underline{ny1}, \text{C}(\underline{ny2}, \text{NIL}))$ is not generalized. The same happens for $\text{C}(\underline{nx1}, \text{C}(\underline{nx2}, \text{NIL}))$. Again (E5) is correctly no more generalized.

8. COMPARISON WITH RELATED WORK

The inductive method we have presented has been influenced in many respects by previous research in induction. First of all, the generalization based on proof can be classified as a successive refinement method. We do not need to iterate the process of guessing and refining, because the formal calculus environment provides for a powerful technique (forced symbolic computation) to exactly adjust the initial guessing. The use of the given computation trace to strengthen the theorem, relates to Brown' and Tarnlund's temporal method based on proofs [4].

Secondly, we have had the advantage of being able to draw on the ideas of Boyer and Moore [2] and of Aubin [1], who tackle the problem of generalizing the theorem to be proven by induction. One basic problem is that of distinguishing different occurrences of a variable. Boyer and Moore generalize terms which the involved functions recur on, thus relating generalization to function definition. In their footsteps, Aubin points out the close relationship among generalization, proof by induction and symbolic computation. His method generalizes those variables which an interpreter would first instantiate during symbolic computation. Thus, only the first and fourth occurrences of x are generalized in the theorem $\text{EQL}(\text{A}(x, \text{A}(x, x)), \text{A}(\text{A}(x, x), x))$. We bring the whole computation structure to bear on the problem, and we can capture rather complex relationships, as shown in the example in Section 7. Because of the peculiar role played by the LR function, Aubin's method would incorrectly generalize the first and third occurrence of x in the theorem $\text{EQS}(\text{LR}(\text{LA}(x, x)), \text{LA}(\text{LR}(x), \text{LR}(x)))$.

9. PROPERTY INDUCTION, THEOREM PROVING AND PROGRAM VERIFICATION

Just as one can prove whether a specific induced property does hold, one can prove that the

inductive method itself is guaranteed to induce valid properties only. We have been able to prove this result, with a few limitations [8].

First of all, the induced theorem must have the form $b(t1, t2)$, where b is a boolean function defined according to the schema

$b(T, T') = BCONST1$ $b(c(x, y), T') = BCONST2$
 $b(T, c'(z, w)) = BCONST3$
 $b(c(x, y), c'(z, w)) = b1(b2(x, z), b(y, w))$

$b1(BCONST4, y) = BCONST5$ $b1(BCONST6, y) = y$
 where $BCONSTi$ are boolean constants (note that $BCONST4 \neq BCONST6$), T, T', c, c' type constructors, and $b2$ a boolean function belonging to the class.

The following definition schemata characterize the class of functions occurring in $t1$ and $t2$.

$n(T, y) = u(y)$ $n(C(x, y), z) = C'(p(x), n(y, z))$
 $r(T, y) = w(y)$ $r(C(x, y), z) = n'(r(y, z), C'(x, T'))$
 $n'(T', y) = v(y)$ $n'(C'(x, y), z) = C''(x, n'(y, z))$

$s(x, y) = f(g(x, y), h(x, y))$

where u, v, w, p and f, g, h either are constructors or belong to the class (note that n' is a restricted n -function).

The limitations on the form of the property and of the boolean functions are introduced to avoid theorems whose proofs need case analysis. The form of the involved functions is constrained so that the structure of their computation gradually changes when the structure of the data terms is gradually changed. Significant functions, predicates and theorems fall into the class. It is certainly not possible to express significance by a figure, but it may be interesting that about 35% of the theorems listed in [1] and involving single reflexive data types are induced by our system and belong to the above class. Classical examples are $EQL(A(x, A(y, z)), A(A(x, y), z))$, $EQL(R(R(x)), x)$, and $EQL(R(A(x, y)), A(R(y), R(x)))$. The reflexivity theorems for EQN , EQL , $EQLN$, and EQS fall into the class and are induced by the system.

At the present, the application of such inductive method to theorem proving has one advantage but suffers from a few limitations. The advantage is that no combinatorial explosion arises in the proof. Free variable introduction is indeed a non-deterministic process, but no nesting of non-deterministic choices is involved. On the other hand, the major limitation of

the proposed method is that it is not proven complete. Thus, it can be used only as an auxiliary tool, which may fail, but at least requiring a bounded amount of resources. Moreover, the method can prove only function properties whose restrictions have been described above. These are heavy limitations for a general purpose theorem prover.

In the framework of program verification, where the goal is in general that of proving properties of functions, it is more likely that the inductive method can help, provided that the theorem to be proven (or a subgoal generated during the proof) is within its reach.

The present inductive method has been implemented into an experimental system mainly by modifying existing TTEL symbolic interpreters. No care has been taken to obtain an efficient system.

APPENDIX I

$(TYPE(EQLN(L, L)) = B; \{EQLN(NIL, NIL) = TT;$
 $EQLN(C(x, y), NIL) = FF; EQLN(NIL, C(z, w)) = FF;$
 $EQLN(C(x, y), C(z, w)) = EQLN(y, w)\})$
 $(TYPE(EQL(L, L)) = B; \{EQL(NIL, NIL) = TT;$
 $EQL(C(x, y), NIL) = FF; EQL(NIL, C(z, w)) = FF;$
 $EQL(C(x, y), C(z, w)) = AND(EQN(x, z), EQL(y, w))\})$
 $(TYPE(EQS(LL, LL)) = B; \{EQS(LNIL, LNIL) = TT;$
 $EQS(LC(x, y), LNIL) = FF; EQS(LNIL, LC(z, w)) = FF;$
 $EQS(LC(x, y), LC(z, w)) = AND(EQLN(x, z), EQS(y, w))\})$
 $(TYPE(AND(B, B)) = B;$
 $\{AND(TT, TT) = TT; AND(TT, FF) = FF;$
 $AND(FF, FF) = FF; AND(FF, TT) = FF\})$
 $(TYPE(PLUS(N, N)) = N;$
 $\{PLUS(0, y) = y; PLUS(S(x), y) = S(PLUS(x, y))\})$
 $(TYPE(HANOI(N)) = N; \{HANOI(0) = 0;$
 $HANOI(S(x)) = S(TIMES(S(S(0)), HANOI(x))\})$
 $(TYPE(TIMES(N, N)) = N; \{TIMES(0, y) = 0;$
 $TIMES(s(x), y) = PLUS(TIMES(x, y), y)\})$
 $(TYPE(A(L, L)) = L;$
 $\{A(NIL, z) = z; A(C(x, y), z) = C(x, A(y, z))\})$
 $(TYPE(R(L)) = L;$
 $\{R(NIL) = NIL; R(C(x, y)) = A(R(y), C(x, NIL))\})$
 $(TYPE(LA(LL, LL)) = LL;$
 $\{LA(LNIL, z) = z; LA(LC(x, y), z) = LC(x, LA(y, z))\})$
 $(TYPE(LR(LL)) = LL; \{LR(LNIL) = LNIL;$
 $LR(LC(x, y)) = LA(LR(y), LC(x, LNIL))\})$

APPENDIX II

We report here the norm computation for some functions given in Appendix I. We look at function definitions in terms of the schema

$f(T,y)=g(y)$ $f(c(x,y),z)=h(x,f(y,z))$.
 If $\text{Norm}(h)$ has the form $a+p1$, then
 $\text{Norm}(f)=\text{Norm}(g)+a*p1$. For example, since
 $\text{Norm}(S)=1+p1$ and $\text{Norm}(C)=1+p1$, then
 $\text{Norm}(PLUS)=\text{Norm}(y)+1*p1=p2+p1$, and $\text{Norm}(R)=$
 $=\text{Norm}(NIL)+\text{Norm}(C(x,NIL))*p1=0+1*p1=p1$. If
 $\text{Norm}(h)=a+b*p1$, then $\text{Norm}(f)=a*(b\uparrow p1)/(b-1)+$
 $+\text{Norm}(g)*b\uparrow p1$. For example, $\text{Norm}(HANOI)=1*(2\uparrow p1-$
 $-1)/(2-1)+0*2\uparrow p1=2\uparrow p1-1$, since $\text{Norm}(g)=0$, $a=1$,
 $b=2$.

REFERENCES

- [1] Aubin, R. "Mechanizing structural induction" Ph.D. Th., U. of Edinburgh, Dept. of Artif. Intel. (1976).
- [2] Boyer, R.S., and Moore, J. S. "Proving theorems about LISP functions". J. ACM 22 (1975), 129-144.
- [3] Brotz, D.K. "Embedding heuristic problem solving methods in a mechanical theorem prover". STAN-CS-74-446, Comp. Science Dept., Stanford Univ. 1974.
- [4] Brown, F.M., and Tarnlund, S. "Inductive reasoning in mathematics". Proc. IJCAI-7. Cambridge, MIT, Mass., August, 1977, pp. 844-850.
- [5] Buchanan, B.G. "Logic of scientific discovery". STAN-CS-66-46. Dept. of Comp. Science. Stanford Univ., 1966.
- [6] Buchanan, B.G., Feigenbaum, E.A., and Lederberg, J. "A heuristic programming study of theory formation in science". Proc. IJCAI-71, London, IC, September, 1971, pp. 40-48.
- [7] Burstall, R.M. "Proving properties of programs by structural induction". Comp. J. 12, 1 (1969), 41-48.
- [8] Degano, P., and Sirovich, F. "Inductive generalization and proofs of function properties". To appear in Comput. Linguist. Comput. Lang.
- [9] Evans, T.G. "A program for the solution of geometric-analogy intelligence test questions". In Semantic Information Processing, M. Minsky Ed., MIT Press, 1968, pp. 271-656.
- [10] Feldman, J., Gips, J., Horning, J.J., and Reeder, S. "Grammatical complexity and inference". STAN-CS-69-125, Comp. Science Dept., Stanford Univ., 1969.
- [11] Hoare, C.A.R. "Recursive data structures". Int. J. Comp. and Inf. Sc. 4, 2 (1975), 105-162.
- [12] Lenat, D.B. "Automatic theory formation in mathematics". Proc. IJCAI-77 Cambridge, MIT, Mass., August, 1977, pp. 866-842.
- [13] Levi, G. and Sirovich, F. "Proving program properties, symbolic evaluation and logical procedural semantics". Lecture Notes in Computer Science, Vol. 62. MFCS, Springer, Berlin, 1975, pp. 569-574.
- [14] McCarthy, J., and Hayes, P. "Some philosophical problems from the standpoint of artificial intelligence". In M.I. 4, B. Meltzer, and D. Michie Eds., Edinburgh Univ. Press, 1969, pp. 466-502.
- [15] Meltzer, B. "Power amplification for theorem-provers". In M.I. 5, B. Meltzer, and D. Michie Eds., Edinburgh Univ. Press, 1969, pp. 165-179.
- [16] Michalski, R.S. "A system of programs for computer-aided induction: a summary". Proc. IJCAI-77 Cambridge, MIT, Mass., 1977, pp. 619-620.
- [17] Newell, A. "Heuristic programming: ill-structured problems". In Progress in Operational Research, III, Aronofsky J.S. Ed., Wiley & Sons, 1969, pp. 662-415.
- [18] Plotkin, G.D. "A further note on inductive generalization". In M.I. 6, B. Meltzer, and D. Michie Eds., Edinburgh Univ. Press, 1971, pp. 101-124.
- [19] Popplestone, R.J. "An Experiment in automatic induction". In M.I. 5, B. Meltzer, and D. Michie Eds., Edinburgh Univ. Press, 1969, pp. 206-215.
- [20] Simon, H.A., and Kotovsky, K. "Human acquisition of concepts for sequential patterns". Psychol. Rev., 70 (1966), 564-546.
- [21] Vere, S.A. "Induction of concepts in the predicate calculus". Proc. IJCAI-75 Tbilisi, GSSR, August, 1975, pp. 281-287.
- [22] Waldinger, R.J., and Lee, R.C.T. "PROW: a step toward automatic program writing". Proc. IJCAI-69 Washington, DC, 1969, pp. 241-252.