# Using theorem provers for PL1EQ as inductive provers

Matthias Fuchs
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
6750 Kaiserslautern
Germany
E-mail: fuchs@informatik.uni-kl.de

December 1992

**Abstract**

This report presents a method to create an inductive proof system by using a theorem prover for PL1EQ (first order logic with equality) as a basic system.
This method not only comprises the general principles necessary for making a theorem prover for PL1EQ capable of performing inductive proofs, but also includes further features that may be added to systems for inductive proofs in order to guide the individual proof (for instance heuristics aimed at optimizing the application of the inference rules with respect to the fact that a proof by induction is to be found).
An emphasis lies on the automatic generation of inductive lemmata which are crucial to the success of inductive proofs. In this domain a range of heuristics were conceived, partly as derivations of ideas of other authors, partly as own developments, which in many cases can generate a lemma that ends the proof attempt successfully. (This includes that this lemma itself can be proved by induction as well.)

## 0. Introduction

Proofs in inductive theories face a wide range of application, not only in mathematics, but also in computer science (here especially in the domain of program verification). Therefore the need for systems for inductive proofs is understandable. The goal of this report is to present a method that represents a general means to convert any theorem prover for PL1EQ into an inductive prover.

Since inductive theories are much harder than the “usual” theories for PL1 or PL1EQ, in the early years of automatic proving a lot of work was invested in the construction of systems that were able to prove in theories concerning PL1EQ. Because these systems nowadays are considerably powerful on account of continuous optimizations, improvements and extensions, it seems to be recommendable to use these systems for inductive proofs by adding components suitable for this purpose.
This is of course only one possible approach to the enormous challenge of inductive proofs. Another approach would be the use of unfailing completion techniques combined with the concept

of proofs by consistency. But this approach involves the development of systems completely different from what is normally considered as theorem provers for PL1EQ. For this reason the proceedings to achieve the extension of an already existing theorem prover for PL1EQ, so that it will be able to perform inductive proofs, must be different and shall be presented in the following paragraph in form of a general overview on the sections of this report:
First of all, a description of the general method for inductive proving, the so-called structural induction, will be given. In this section 1 it will also be explained how that method and the underlying theorem prover for PL1EQ relate to form a device for accomplishing inductive proofs. Furthermore the inductive theory, in which the thereby created system is able to prove, will be defined.
Section 2 will describe some features which can be very useful for guiding an individual inductive proof. It includes heuristics for the application of inference rules as well as heuristics related to the problem of inductive lemmata in general, for instance criteria which allow to detect the need for inductive lemmata.
The problem of generating automatically the inductive lemmata, which are necessary for a successful proof, will be covered in section 3. The general idea behind the method for the automatic generation of inductive lemmata consists in the manipulation of a subset of the formulas inferred in the course of an inductive proof attempt. This subset will be called “lemma-candidates”. The manipulation mainly consists in transformations of syntactical nature, realized by various heuristics. Therefore the process of automatic lemma-generation is divided into two steps. In the first step (section 3.2), a subset of formulas (i.e. the lemma-candidates) is extracted. This subset is subsequently reduced (because our aim should be the generation of just one useful lemma). This will be discussed in sections 3.2.1 and 3.2.2. After that, the remaining formulas can be manipulated. This task is taken care of in the second step (section 3.3). A number of heuristics for this purpose will be presented and illustrated by examples (sections 3.3.1 through 3.3.6). How these various heuristics interact so as to get closer to the ideal goal of creating exactly one lemma is the subject of section 3.4. Moreover a concept will be introduced which allows us to classify these heuristics according to the effects the lemmata generated by them have on the respective proof.
The last section 4 will summarize the essential parts of this report, point out some limits of the presented methods and give a brief outlook on what could be further interesting and useful items in this domain of research.

## 1. Structural Induction

In this section we shall come to know the proof principle which represents the framework for our inductive prover. But before we can come to the heart of the matter, we have to introduce some preliminary definitions concerning the notions term, formula, substitution etc.

Definition 1.1: *term*
Let $FS=\{f_1,..,f_k\}$ be a set of function-symbols with arity $\tau(f_i)>0$, where $k\geq 0$, $C=\{c_1,..,c_m\}$ be a set of constant-symbols, where $m>0$, and V be a (enumerable) set of variables.

t is in **term(FS,C,V)** iff
(a) $t \in C$ or
(b) $t \in V$ or
(c) $t_1,..,t_n \in$ term(FS,C,V), $f \in FS$, $\tau(f)=n$, $t \equiv f(t_1,..,t_n)$.
(Note: If in (c) n=2, then we may prefer infix notation.)

If there is no ambiguity, we shall use the short version “t is a term” instead of “t is in term(FS,C,V)”. If V is empty, then term(FS,C) denotes the set of ground-terms.

Definition 1.2: *formulas*
$\forall, \exists$ denote the quantifiers as usual, $\neg$ is the negation symbol, PS a finite set of predicate-symbols and OP a finite set of binary logic operators ($OP=\{\wedge,\vee,..\}$), $x \in V$.
F is called a **formula** iff
(a) $F \equiv Q\ x\ (F')$, F’ formula, $Q \in \{\forall,\exists\}$ or
(b) $F \equiv \neg(F')$, F’ formula or
(c) $F \equiv (F_1\ op\ F_2)$, $F_1$, $F_2$ formulas, $op \in OP$ or
(d) $F \equiv P(t_1,..,t_n)$, $t_i \in$ term(FS,C,V), $n \geq 0$, $P \in PS$.
If $F \equiv P(t_1,..,t_n)$ or $F \equiv \neg P(t_1,..,t_n)$, then F is called a literal.
(Note: If in (d) n=2, then we may prefer infix notation.)

Since we can look upon formulas as terms with a special structure (e.g.$\forall$ x (F) could be interpreted as ALL(x,F)), both notions will be used synonymously if there is no risk of confusion. Moreover, any formula F can be written without quantification (implicitly $\forall$-quantified), e.g. utilizing the so-called clause-normalform ([Ni80]). To avoid notation-fiddling, we assume for every formula to be represented this way unless stated otherwise.
Later in this report it will become necessary to pinpoint positions in terms and terms occuring at these positions. This is facilitated by the following definition.

Definition 1.3:
Let t be in term(FS,C,V).
O(t) is called the set of positions resp. **occurences** in t, and it is
$O(t)=\{\varepsilon\}$ if t is a constant or variable; $t|\varepsilon \equiv t$;
$O(t)=\{ip \mid p \in O(t_i), 1 \leq i \leq n\} \cup \{\varepsilon\}$ if $t \equiv f(t_1,..,t_n)$ and $t|ip \equiv t_i|p$; $t|\varepsilon \equiv t$;
If $p \in O(t)$, then $s \equiv t|p$ is called a subterm of t.
If $p \neq \varepsilon$, then s is called a proper subterm of t.
Furthermore, if t’ is a term, then $t[p \leftarrow t']$ denotes the term obtained when replacing the term t|p at position p in t by t’.

Further definitions we shall need for defining the inductive theory we want to be able to prove in are listed in the sequel.

Definition 1.4:
FS, C and V are as introduced in definition 1.1, V’ be a finite subset of V.

(a) An endomorphism $\sigma$:term(FS,C,V)$\rightarrow$term(FS,C,V) is called a **substitution** iff
$\sigma(x)\in$ term(FS,C,V) for all $x\in V'$, $\sigma(z)=z$ for all $z\in V\text{-}V'$, $\sigma(c)=c$ for all $c\in C$ and $\sigma(f(t_1,..,t_n))=f(\sigma(t_1),..,\sigma(t_n))$ for all $f\in FS$, $t_i\in$ term(FS,C,V), $\tau(f)=n$.

(b) $\sigma$ is called a **ground-substitution** iff
$\sigma$ is a substitution where $\sigma(x)\in$ term(FS,C) (i.e. $\sigma(x)$ is a ground-term) for all $x\in V'$, $\sigma(z)=z$ for all $z\in V\text{-}V'$.

The inductive theory ITh naturally depends on the set Ax of axioms and the inference relation $\vdash$ of the underlying prover for PL1EQ. Hence we define ITh(Ax)={F | Ax $\vdash \sigma$(F) for all ground-substitutions $\sigma$}. This definition reflects the idea of using the data-model which is actually represented by term(FS,C).

Note:
With the determination of Ax we also implicitly determine FS, C and PS such that
FS={f | f function-symbol in F$\in$ Ax},
C ={c | c constant-symbol in F$\in$ Ax},
PS={P | P predicate-symbol in F$\in$ Ax}.

Mostly, there is no need to take into account all ground-substitutions, considering only the so-called constructor-ground-substitutions.

Definition 1.5:
Let $FS_C\subseteq FS$, $C_C\subseteq C$.
If for any term $t\in$ term(FS,C) there is a $t'\in$ term($FS_C$,$C_C$) so that Ax $\vdash$ t=t' (we also write $t=_{Ax}t'$ for simplicity), then $FS_C$ is called a set of constructor-function-symbols, $C_C$ is called a set of constructor-constant-symbols.
The set $FS_C\cup C_C$ is a set of **constructors**.
$g\in FS_C\cup C_C$ is called a constructor.
$t\in$ term($FS_C$,$C_C$) is a constructor-groundterm.
We call $\sigma$ a constructor-ground-substitution, if $\sigma$ is a (ground-) substitution, where $\sigma(x)\in$ term($FS_C$,$C_C$) for all $x\in V'$ (V' is a finite subset of the set V of all variables), $\sigma(z)=z$ for all $z\in V\text{-}V'$.

Hence we can redefine ITh(Ax):

Definition 1.6:
Let Ax be the set of axioms, $\vdash$ the inference relation of the underlying prover for PL1EQ.
ITh(Ax)={F | Ax $\vdash \sigma$(F) for all constructor-ground-substitution $\sigma$}
Any formula F$\in$ ITh(Ax) is called an inductive theorem. (If its membership in ITh(Ax) is not (yet) confirmed, it will be denoted inductive theorem nevertheless.)

*Example*: (constructors and inductive theory)
Ax={x+0=x, x+s(y)=s(x+y)}.

Consequently, we have FS={+,s} and C={0}.

assertion: {s,0} is a set of constructors. (Of course {+,s,0} is also a set of constructors. But we are interested in smaller sets because we want to reduce effort.)

proof: (Noetherian induction)
Let $s,t \in term(FS,C)$; s>'t :iff $|l_s|>|l_t|$, where > represents the usual "greater-than"-relation on natural numbers and $l_u=\{p \in O(u) \mid u|p=s_1+s_2,\ s_1,s_2 \in term(FS,C)\}$ for any $u \in term(FS,C)$.
Then >' is Noetherian.
$P(t):=\exists t' \in term(FS_C,C_C): t=_{Ax} t'$;
If $|l_t|=0$, then P(t) holds with $t' \equiv t$.
Let $|l_t|>0$, P(s) hold for all $s \in term(FS,C)$ with t>'s. Hence there is a $p' \in O(t)$ with $t|p'=s_1'+s_2'$. Choose $p \in O(t)$ with $t|p=s_1+s_2$ so that $s_1 \equiv s^k(0)$, $s_2 \equiv s^j(0)$, $k,j \geq 0$. Then $t|p=_{Ax} s^{k+j}(0)$, therefore $t=_{Ax} t[p \leftarrow s^{k+j}(0)]=:t'$. So $|l_t|>|l_{t'}|$, and consequently t>'t'. According to the induction assumption, P(t') holds. Therefore there is a $u \in term(FS_C,C_C)$ with $t'=_{Ax} u$. Thus $t=_{Ax} u$ and P(t). ❒

This set of constructors is now used to prove that the commutativity of + is an inductive theorem.

assertion: $x+y=y+x \in ITh(Ax)$

proof:
Let σ be an arbitrary constructor-ground-substitution, i.e. $\sigma(z) \in \{s^k(0) \mid k \in \{0,1,..\}\}$ for a finite subset V' of all variables. Thus $\sigma(x+y) = \sigma(x)+\sigma(y) = s^i(0)+s^j(0) =_{Ax} s^{i+j}(0) \equiv s^{j+i}(0) =_{Ax} s^j(0)+s^i(0) = \sigma(y)+\sigma(x) = \sigma(y+x)$ for all constructor-ground-substitutions σ. ❒

In general there will be infinitely many constructor-ground-substitutions, so that testing $Ax \vdash \sigma(F)$ for all constructor-ground-substitutions σ to prove $F \in ITh(Ax)$ in compliance with the definition of ITh(Ax) is out of the question.
For an algorithmic realization we have to find another way. Using the so-called "structural induction" is a possible approach since it reduces the infinite number of tests to a finite number, relying on the following principle:

*structural induction*
Let F be a formula, x a variable in F (the induction variable), Ax a set of axioms, and $F[x \leftarrow t]$ denote the formula obtained by replacing all occurences of x by t, where $t \in term(FS_C,C_C)$.
F is in ITh(Ax) if (a) and (b) hold:
(a) $Ax \vdash F[x \leftarrow c]$ for all $c \in C_C$ : BASE-CASES
(b) $Ax \cup \{F[x \leftarrow t_i] \mid 1 \leq i \leq n\} \vdash F[x \leftarrow f(t_1,..,t_n)]$ for all $f \in FS_C$ with $\tau(f)=n$ : INDUCTION-STEPS
(The $t_i$'s represent arbitrary constructor-groundterms. The formulas $F[x \leftarrow t_i]$ are called hypotheses. The formula generated in a base-case or induction-step is called conclusion.)

proof:
Since $Ax \vdash F[x \leftarrow t]$ for all $t \in term(FS_C,C_C)$ implies $Ax \vdash \sigma(F)$ for all constructor-ground-

substitutions $\sigma$, it suffices to prove the former statement.
Noetherian induction: $P(z):=Ax \vdash F[x \leftarrow z]$; $s>t$ :iff t is a proper subterm of s (thus > is Noetherian). Because of (a) P(z) holds for all $z \in C_C$. Let $z \in term(FS_C,C_C)$, $z \notin C_C$, and for all y with $z>y$ P(y) hold. Therefore $z \equiv f(t_1,..,t_n)$ for some $t_1,..,t_n \in term(FS_C,C_C)$, $f \in FS_C$, and since $z>t_i$ for all $i \in \{1,..,n\}$ $P(t_i)$ holds for all $i \in \{1,..,n\}$. Hence $Ax \cup \{F[x \leftarrow t_i] \mid 1 \le i \le n\} \vdash F[x \leftarrow f(t_1,..,t_n)]$ if and only if $Ax \vdash F[x \leftarrow f(t_1,..,t_n)]$. With (b) we have P(z). ❐

Remarks:
Some of the hypotheses may be omitted if we dispose of sort information. For the same reason, parts of the base-cases and induction-steps may be skipped. (The lack of sort information can be compensated by selecting subsets of $FS_C$ and $C_C$ so that all terms generable comply with the implicitly given sort of the induction variable x.)

Equipped with this method, it is easy to write down an algorithm enabling us to perform inductive proofs (assuming that the underlying prover for PL1EQ - represented by $\vdash$ - is already available or is not considered a major difficulty):
(1) for all $c \in C_C$: prove $Ax \vdash F[x \leftarrow c]$;
(2) for all $f \in FS_C$ : prove $Ax \cup \{F[x \leftarrow t_i] \mid 1 \le i \le n, t_i \text{ new constant symbol}\} \vdash F[x \leftarrow f(t_1,..,t_n)]$;

It is apparent that (1) and (2) realize in a straight forward way (a) and (b) we encountered when introducing structural induction. Hence, whenever (1) and (2) could be executed successfully (assuming correctness of $\vdash$), we have proved $F \in ITh(Ax)$.
(Note: Later in this report, if no ambiguity can arise, we shall sometimes use expressions like “proof is completed”, though only referring to the proof of a base-case or induction-step, meaning that the proof of the respective base-case or induction-step has been completed.)
Unfortunately, ITh(Ax) is not even semi-decidable. Therefore, we cannot prove for every $F \in ITh(Ax)$ to have this property merely by applying the steps (1) and (2).

*Example*:
We already know that $x+y=y+x$ is in $ITh(\{x+0=x, x+s(y)=s(x+y)\})$. But when making use of the above procedure, we have the base-case $0+x=x+0$ (using y as induction variable, what does not matter because of the symmetrie of the problem). Simplification yields $0+x=0$, and we cannot proceed any further.

The dilemma is the need for additional inductive theorems, called lemmata, which must be added to the set of axioms in order to be able to complete a proof. (In the example, the lemma $0+x=x$ would be satisfactory for the completion of the base-case.) Since the lemmata we have to use are inductive theorems, no prover for PL1EQ can find them. So they must in the simplest case be provided by the user and be proved separately by induction as well. But apart from this, heuristics can be incorporated whose jobs consist in generating the lemmata necessary for a successful proof, as we shall see in a later section.

## 2. Additional features

In this section a range of methods will be introduced which all serve the purpose of more and more automatizing the inductive prover outlined in the preceding section. The automatic generation of inductive lemmata, which also belongs to these methods, will be handled in a separate section because of its outstanding importance.

The following subjects will be covered:
2.1. automatizing the choice of the induction variable
2.2. induction on several variables
2.3. heuristics for guiding the inference rules with respect to the context of an inductive proof
2.4. detecting the need for inductive lemmata

### 2.1. Automatizing the search for the induction variable

The proper choice of the induction variable plays an important role in an attempt to prove some theorem by induction as we shall see through the following example.

*Example*:
The set of axioms be Ax={x+0=x, x+s(y)=s(x+y)}.
When attempting to prove x+(y+z)=(x+y)+z$\in$ITh(Ax), choosing x as the induction variable, we cannot complete the proof without the lemma s(x)+y=s(x+y). But if we use z as induction variable, there are no problems.

Leaving this choice to the user is a respectable but not very satisfactory solution of this problem. Therefore a rather simple but quite effective heuristic was developed which suggests a variable for the use as the induction variable. This heuristic will now be introduced: We already know that both in the base case and in the induction-step there are instances of the theorem to be proved. Those instances are in principle generated by replacing all occurences of the induction variable by a term with a constructor function symbol as top-level symbol. For a successful proof, it is necessary to manipulate the respective instance in a “proper” way, so that either axioms or hypotheses can be applied. This manipulation and the applicability of axioms or hypotheses mainly depend on the premise that the term, which was substituted for the induction variable when creating an instance, occurs at positions that potentially allow an application of axioms. These positions (they will be called *potential induction places* from now on) are closely related to the argument positions which are responsible for recursive definition. Hence we have to concentrate on the potential induction places of all functions and predicates occuring in the theorem. The idea is to count for every variable in the theorem the frequency of occurences at such positions. That variable which in this way can achieve the highest number is suggested.

*Example*:
Ax={x+0=x,x+s(y)=s(x+y)}; + is recursively defined in the second argument; for the theorem x+(y+z)=(x+y)+z we obtain the numbers 0,1 and 2 for the variables x,y and z respectively. Therefore z is suggested.

## 2.2. Induction on several variables

The structural induction as it was introduced in section 1 heavily depends on the availability of suitable inductive lemmata. But sometimes it is recommendable to use another technique to complete an inductive proof: the so-called induction on several variables. This shall be illustrated by an example:

*Example*:
Ax={0≤x, ¬(s(x)≤0), s(x)≤s(y)↔x≤y};
Theorem: x≤y∨y≤x
Induction variable: x (y would be as good a choice because of symmetrie)
base-case: 0≤y∨y≤0 is ok.
hypothesis: t≤y∨y≤t
induction-step: s(t)≤y∨y≤s(t) cannot be proved with the set of axioms.
Instead of using and of course having to prove the necessary and sufficient “special purpose” lemma (t≤y∨y≤t)→(s(t)≤y∨y≤s(t)) (“special purpose” because of the special constant t in the potential lemma) the induction-step is proved by a further induction on the (remaining) variable y, in the course of which the hypotheses of all former inductions are still available (here there is only one hypothesis):
base case': s(t)≤0∨0≤s(t) is ok.
hypothesis': s(t)≤t'∨t'≤s(t)
induction-step': s(t)≤s(t')∨s(t')≤s(t) ⇒ t≤t'∨t'≤t is ok (hypothesis)

Induction on several variables is not at all a mandatory feature for an inductive prover, but it sometimes helps to keep the number of lemmata reasonably low and thereby serves the purpose of achieving better transparency.

## 2.3. Guiding the inference rules with respect to the context of an inductive proof

Whenever talking about automatic proving the subject of guiding the inference rules of any proving system so as to obtain higher efficiency is very important. A commonly accepted and in a lot of implementations successfully used heuristic is the set of support (SOS) - strategy (see [CL73]). For our inductive prover this principle will be slightly extended. Instead of one SOS we have in fact three of them: one for the conclusion (i.e. the instance created in the base case or the induction-step), one for the hypotheses and the third for further formulas (for example lemmata) which are considered to be of importance in the course of the proof. All formulas which are not in one of these three sets are referred to as the remaining formulas. By literally trebling the SOS, what has become possible through the knowledge provided by the context of an inductive proof, a more sophisticated hierarchy for the application of the inference rules can be established. While using the usual SOS-strategy we just have formulas belonging to the SOS and formulas not belonging to the SOS, we here have the means of further differentiation: since the conclusion plays the role of the goal in an inductive proof, inferences involving a formula connected to the conclusion should be preferred; inferences involving formulas connected to one of the hypotheses should be considered to be second most important because the hypotheses also play a deciding role in an inductive proof; the formulas of the third set and the remaining formulas can be handled according

to the usual SOS-strategy. Moreover the inferences involving certain combinations (for instance “conclusion and hypothesis”) can be rated in a differentiated way, where rating is done on a heuristical basis, of course. Analogously to the SOS-strategy, inferences involving formulas of the same set or partition (for example “conclusion and conclusion”) should be delayed as much as possible.
Certainly, this extended SOS-strategy is only a small step towards finally creating a “real efficient” control mechanism for the inference rules. It should be regarded as just another possibility to develop a heuristic for inference rule application whose performance clearly depends on how well it can be integrated in the individual case.

**2.4. Detecting the need for inductive lemmata**

The problem of inductive proofs requiring inductive lemmata has already been discussed, but so far nothing was mentioned about how to get the idea during a proof that an inductive lemma might be missing. Therefore this section will present two methods (i.e. heuristics) for this purpose.

2.4.1. In section 2.3, inferences involving the conclusion were said to be most promissing because the conclusion is in fact the goal of the attempted inductive proof. So, if it is discovered during a proof that those inferences which are considered to be essential for the success of the attempt no longer involve a conclusion (or a derivate thereof), it is probable that inductive lemmata might be needed. The definition of the term “essential inference” very much depends on the kind of basic proving system and must be determined in the individual case.

*Example*:
If a theorem prover for PL1EQ is used that works according to the resolution principle with paramodulation (and factorization), and a theorem formulated in pure PL1 is to be proved, resolution should be regarded as the essential inference, not paramodulation.

2.4.2. Another hint is the “inflation of formulas”, i.e. a repeated, recursive structure of formulas created in the course of the proof attempt.

*Examples* (inflated formulas):
s(s(s(s(x)))), f(x,g(f(y,g(f(z,a))))).

This might happen through the repeated application of a sequence of inferences to a formula and the thereby created formulas. Whenever this occurs, it is rather obvious that the proof procedure has run into a “dead end” (provided that this phenomenon is not an inherent fault of the basic proving system) and is in a dire need for inductive lemmata to help it out of it.

*Example*:
Ax={x+0=x,x+s(y)=s(x+y),even(0),¬even(s(0)),even(x)↔even(s(s(x)))}
Trying to prove even(x+x) leads in the induction-step to:
hypothesis: even(t+t)
induction-step: even(s(t)+s(t)) ⇒ even(s(s(t)+t))

With the given axioms the prover can't help creating formulas of the form even($s^{2i}$(s(s(t)+t))), i≥0, which show the symptom of inflation. And actually, an inductive lemma, i.e. s(x)+y=s(x+y), can break the vicious circle.

Hence adding a component to our inductive prover which at least warns the user if an inflation exceeding a given threshold occurs in any formula created represents indeed a useful and powerful method to detect the need for inductive lemmata.
But first of all, the concept of inflation must be properly defined. We have to give a concrete meaning to the notion "inflation", thus enabling us to "measure" in some way the inflation of any given formula (not only for the simple cases like s(s(s(x))), but also for complex structures like $f(x_1,g(f(x_2,g(f(x_3,a,x_3)),x_2)),x_1)$ ). The realization of all this is quite extensive and is therefore listed in appendix B for those readers who are interested in details.

The section shall be concluded with a remark concerning a rather obvious fact: If at some point of the proof-process no further inferences are possible, it is clear that inductive lemmata are missing (assuming correct axiomatization and validity of the theorem to be proved).

## 3. Automatic lemma-generation

### 3.1. Motivation

We already mentioned in previous sections the essential importance of inductive lemmata for the success of an inductive proof. This necessity of inductive lemmata can be explained by the fact that the lemmata needed are "genuine" inductive lemmata (i.e. they are not valid in all models of the respective set of axioms) and that they can therefore not be found or inferred by a theorem prover for PL1EQ. Thus the availability of appropriate inductive lemmata is mandatory (as opposed to the optional availability of lemmata for proofs in PL1EQ merely because of speed up reasons). As our aim consists in constructing a theorem prover which be automatized as much as possible, the convenient, but not very satisfactory alternative of providing manually the needed lemmata (i.e. by the user), should not be acceptable. Hence we have to take a look at the issues of automatic lemma-generation.
It is clear that - due to the undecidability of inductive theories - automatic lemma-generation can only be performed on a heuristical basis.
The first question that arises is the selection of the appropriate moment for the intended lemma-generation. Basically, there are two possibilities:
(1) lemmata may be generated before the proof is started, using only the knowledge provided by the given axioms;
(2) lemmata may be generated during an interrupt of a proof attempt, using all the knowledge that has been inferred so far;

Since most of the time the need for inductive lemmata and the idea how they should look like becomes more obvious after a number of inferences, we shall concentrate on alternative (2). But

alternative (1) is nevertheless worth thinking about and may possess interesting abilities.
There already exists a variety of heuristics dealing with automatic lemma-generation. They mostly carry through syntactical manipulations of some formulas. While it is not always clear where these formulas come from, in our case the formulas representing the state of the current proof attempt can play this role. But it does not make much sense to take into account all formulas. The formulas used in the base-case or induction-step are instances of the theorem to be proved. As such they have the position of a goal (and are referred to as conclusion). Consequently these and the formulas inferred with their participation (which will be called conclusion-formulas) are top-candidates when it comes to lemma-generation. Now it becomes clear that the process of automatic lemma-generation will be performed in two steps:
First of all, the conclusion-formulas must be extracted from the set of formulas representing the state of the current proof. The extracted formulas then act as so-called lemma-candidates after a slight modification: If the proof is being led for an induction-step, then the conclusion-formulas might contain constant symbols introduced through the hypotheses. In order to be able to generate more general lemmata and because of the fact that those constant symbols represent arbitrary ground-terms, they are substituted by (new) variables.
Since there will usually be more than one lemma-candidate and since we intend to produce in the ideal case exactly one lemma, this first step is combined with a reduction of the set of lemma-candidates obtained so far. The way this can be achieved will be explained later in this section (see 3.2.1 and 3.2.2).
In the second step the remaining lemma-candidates undergo the already mentioned (syntactical) manipulations by various heuristics, thus generating one (in the ideal case) or more lemmata. Some of these lemmata may not at all be inductive lemmata, because they were generated by heuristics. To avoid the introduction of a notion like “potential lemma”, we shall call them “lemma” nevertheless. In this context it might be interesting to think about methods for validating lemmata created by heuristics ([Pr92]).
The following two subsections are devoted to the more detailed description of these two steps of automatic lemma-generation.

## 3.2. Step One. Creating and confining the set of lemma-candidates.

Extracting the conclusion-formulas, which can easily be transformed into lemma-candidates in the way outlined above, theoretically imposes hardly problems. We only have to check for every formula available in the current proof whether it stems from the conclusion or not. This can either be achieved by “tracking back” the inference chain which led to the formula in question (what is not a very efficient method), or by associating a flag to every formula, which is propagated during the inference process, indicating the membership to the set of conclusion formulas. A slight problem arises when we want to define which formulas should be considered as stemming from the conclusion. There are certain deductions involving a conclusion-formula which infer formulas that should not be regarded as conclusion-formulas (although a conclusion-formula participated in their generation). This seems to contradict the (informal) definition for conclusion-formulas given above. A motivation for this constraint shall be presented through the following example.

*Example 3.1*:
Suppose we have the two facts
(A) P(f(x,a),x)
(B) f(b,y)=g(y)
which allow to derive (C) P(g(a),b) by paramodulation. If (A) is a conclusion-formula, then it is quite obvious that (C) should be considered as a conclusion-formula as well. If (B) is a conclusion-formula (and (A) is not a conclusion-formula), then it does not seem recommendable to regard (C) as a conclusion-formula.
To justify this way of proceeding for the latter case, let us take a look at the following consideration: If we do not prove by refutation, then this paramodulation is invalid anyway since we used a formula which *is* to be proved. Otherwise it is a legal inference, but making formula (C) a lemma-candidate will most certainly result in an incorrect lemma. One reason is the negation of the initial goal of the proof which we must undo in some way for formula (C). But we cannot do that properly in this case.

This hint towards a first reduction of the set of conclusion-formulas (and thereby also reducing the number of lemma-candidates) must be used cautiously, and the application thereof should be thought over carefully for every individual inference rule.
Furthermore, if the theorem prover for PL1EQ in use works according to the principle “proof by refutation”, then the initial negation of the conclusion must be taken into account properly. The problems arising in this context are not very difficult to handle, if we confine ourselves to ∀-quantified theorems only. (Basically, it suffices under these restricted conditions to replace all skolem-constants by new variables and to negate the conclusion-formulas to revert the initial negation.) As existentially quantified theorems cause tremendous difficulties, we shall omit here theorems involving existential quantification entirely. (An in-depth coverage of this subject can be found in [Bi91].)
But apart from these contemplations, the identification of conclusion-formulas and their extraction combined with the modifications just described (so as to be used as lemma-candidates) is merely an implementational problem.
So let us concentrate our effort to the reduction of the set of lemma-candidates for the purpose of our ideal goal, the generation of exactly one (inductive) lemma. Two heuristics were conceived in order to eliminate a part of the lemma-candidates. They will be described in the sequel.
(Note: A third heuristic is the use of the subsumption-criterion, i.e. preferring those lemmata that subsume other ones to those subsumed by them, e.g. prefer P(x,y) to P(s(x),s(y)). But we shall not consider it here in detail, yet keeping it in mind as a further strategy.)

3.2.1. “Minimal number of different subterms on places suitable for induction variables”
The goal of automatic lemma-generation is the generation of inductive lemmata which have a positive influence on the current proof and which can be proved by induction as well. Otherwise such a lemma would be pointless. This first heuristic for reducing the set of lemma-candidates applies a related idea: it tries to measure (on a heuristical basis) the provability of lemma-candidates. Knowing the importance of the choice of the induction variable, the principle consists in counting the number n of distinct terms occuring at positions in an individual lemma-candidate

which would be suitable positions for the induction variable. “Suitable positions” are again the argument positions on which recursion is performed, i.e. potential induction places. The less the value of n, the better the respective lemma-candidate is estimated. The heuristic therefore discards all lemma-candidates for which it calculates such a number n that is not minimal.
The following example should demonstrate the way this heuristic works and serve as a motivation for the underlying principle:

*Example 3.2*:
Suppose we have a set of axioms
(1) x+0=x
(2) x+s(y)=s(x+y)
and we want to prove the commutativity of +, i.e. x+y=y+x is our theorem. We shall concentrate here on the interesting induction-step: The hypothesis is x+t=t+x (if y is the induction variable, what in this example really doesn’t matter). Hence, we obtain x+s(t)=s(t)+x in the induction-step, which can be rewritten with the help of axiom (2) to s(x+t)=s(t)+x. Applying the hypothesis, we additionally create s(t+x)=s(t)+x. This means that we have at this point of the proof the two conclusion-formulas s(x+t)=s(t)+x and s(t+x)=s(t)+x. They are transformed into lemma-candidates by the already known substitution of t by a (new) variable (z) (see 3.1) and we obtain
(a) s(x+z)=s(z)+x
(b) s(z+x)=s(z)+x
Since argument position 2 of function + is the position where recursion takes place according to the defining axioms, {z,x} and {x} are the terms on that argument positions in the lemma-candidates (a) and (b) respectively. Therefore lemma-candidate (a) is discarded, because our heuristic counts here two distinct terms (x and z) on recursion argument positions, whereas it counts only one for (b). When proving both (a) and (b), we can see that (b) can be proved without difficulties. But for the proof of (a) we need (b), what clearly signalizes that (b) is the better choice.

In general, the presented method sustains the issues inherent to inductive proving which were already discussed in the context of the selection of the induction variable in section 2.1.

Note:
Even in the case of non-variable terms occuring on potential induction places, this method still makes sense. As we shall see later in this section, under certain conditions these non-variable terms may be replaced by variables (“generalization”).

3.2.2. “Avoiding inflated formulas”
Another possibility to reduce the set of lemma-candidates is the elimination of inflated formulas. This notion was already introduced in section 2.4 and describes the symptom of a regular (recursive) structure of a term. So, f(s(s(s(s(x)))),a) is considered to be inflated. The degree to which a formula is inflated is expressed in natural numbers, where high values indicate a high degree of inflation. (How these values are computed is shown in appendix B in detail.) The idea is to remove all lemma-candidates that do not have a minimal inflation degree. Once again for motivation and explanatory reasons, we shall take a look at an example:

*Example 3.3*:
Let the set of axioms be
(1) x+0=x
(2) x+s(y)=s(x+y)
(3) even(0)
(4) ¬even(s(0))
(5) even(x)↔even(s(s(x)))
If we want to prove the inductive theorem even(x+x), we obtain even(s(t)+s(t)) in the induction-step (the base case is trivial). This can be rewritten to even(s(s(t)+t)). Since neither the hypothesis even(t+t) nor any axiom other than axiom (5) can be applied, we get lost in the infinite deduction chain even(s(s(t)+t))↔even(s(s(s(s(t)+t))))↔...
When automatic lemma-generation is invoked, we have in general n lemma-candidates of the form even($s^{2i}$(s(s(t)+t))), $0 \leq i < n$. It is obvious that the lemma-candidates with $i>0$ are redundant. So, their elimination by applying the described criterion of minimal inflation is welcome.

Despite these apparent advantages of this method it must not be denied that it hosts a certain risk of eliminating possibly suitable lemma-candidates. It is therefore in its initial and simple form especially open to further extensions and improvements.

### 3.3. Step two. Generating lemmata.

At this point of the process of automatic lemma-generation we dispose of a (reduced) set of lemma-candidates. As these lemma-candidates merely represent a subset of the formulas available in the current proof, it is usually necessary to submit them to further manipulations in order to obtain lemmata which themselves have a good chance of being provable by induction. For this task a range of heuristics is at our disposal that will be described in this paragraph. But before that, we shall introduce a classification for these heuristics in terms of the effect the lemmata created by them have on the respective current proof. It is clear that the generated lemmata should at least have a positive influence on the proof, i.e. they should bring the proof closer to a successful termination. But besides this general request, a significant number of the heuristics to be presented have the property of providing lemmata that always conclude the proof (of a base-case or induction-step). These lemmata shall from now on be referred to as sufficient lemmata. Moreover, those heuristics that generally produce sufficient lemmata will also be called sufficient.
(Note: If a lemma is sufficient, this does not imply that it is correct, i.e. it is valid in the inductive theory currently investigated through the proof attempt.)
Sufficient lemmata are naturally very convenient, because we do not have to add them to the current set of formulas and to continue the inference procedure; it suffices to focus on the proof of the (sufficient) lemma. Nevertheless there are important heuristics that do not produce sufficient lemmata in general.
It must be noted that every lemma-candidate is sufficient, because it is a generalized derivate of the conclusion that would conclude the proof if added as a lemma. It is important to keep this in mind, because this fact is essential when we want to classify the heuristics for lemma-generation

according to the criterion of sufficiency.
We shall now present heuristics for generating lemmata out of lemma-candidates.
(Note: Almost all heuristics we shall come to know in the following subsections are supposed to work on lemma-candidates though this may not be stated explicitly every time.)

3.3.1. Generalization of common subterms

Generalization is usually understood as a procedure which creates a formula F' from a formula F so that F'$\rightarrow$F holds. (F' is said to be more general.) Although this method complicates the proof (or even makes it impossible) for theorems in PL1EQ, an inductive proof can benefit thereof. A reason for this phenomenon is the fact that - at least in the induction-step - the hypotheses must be applicable to the instance created in the induction-step in order to complete the proof. The more general the hypotheses are, the easier this task can be accomplished. As the hypotheses themselves are instances of the theorem and because the lemma we want to generate has to be a theorem as well when an attempt is made to prove it, a generalization offers a good chance to create lemmata that are provable by induction. For these reasons this first heuristic for lemma-generation relies on the generalization of the available lemma-candidates.
Any replacement of a subterm of a formula by a (new) variable is a generalization. But replacing indifferently subterms not only causes a lot of indeterminism, but also increases the danger of generating formulas that are no more in the respective (inductive) theory, i.e. they are over-generalized. Confining the generalization (this means in our case the replacement of subterms by new variables) to so-called common subterms significantly diminishes this negative effect. A common subterm in this context (see also [BM79], [Au77] and [Hu87]) is a subterm of a formula which neither is a constant nor a variable and which occurs either in more than one literal or in distinct sides of an equation.

*Examples 3.4*:
$\forall x:P(x,f(x))\vee Q(f(x))$ - f(x) is a common subterm
$\forall x:f(g(x),a)=g(x)$ - g(x) is a common subterm
$\forall x:P(g(x),g(x))$ - g(x) is not a common subterm

The generalization of common subterms can now be defined as the replacement of all occurences of a common subterm by a new variable. If there exist more than one common subterm in a formula, we have to decide which of them to generalize. We shall discuss this problem later in this section.
Generalizing common subterms of lemma-candidates according to this convention in most cases provides us with lemmata that are not over-generalized. This is of course an experimental and empirical result, sustained by many encouraging tests (see also appendix A).

*Example 3.5*:
From s(x)+s(y)=s(y)+s(x) we obtain after generalizing the common subterms s(x) and s(y) the equation x+y=y+x which can be proved more easily than the original one, using the axioms of example 3.2.

Although the restrictive generalization of common subterms substantially improves reliability, there are still cases in which it may fail.

*Example 3.6*:
Suppose we dispose of a function isort:LIST→LIST (see also appendix A) that computes sorted lists (of natural numbers for instance) w.r.t. some ordering. Then it is clear that isort(isort(x))=isort(x) holds. Generalizing the common subterm isort(x) results in the equation isort(z)=z which is not valid anymore.

In this case the failure of our method can be compensated by the introduction of so-called generalization-lemmata. This notion is described in detail in [Hu87]. We shall therefore only take a glance at the issues of this domain and consider merely the basic principle:
The problem we run into when generalizing the common subterm isort(x) is caused by the circumstance that isort(x) represents sorted lists (not arbitrary lists). If we have a predicate ORD(x) at our disposal which is true if and only if x is a sorted list, then we can code that property of isort by formulating the following implication ORD(z)→isort(z)=z. If we can prove this implication, then the original equation isort(isort(x))=isort(x) is also proved, because ORD(isort(x)) → isort(isort(x))=isort(x) ⇒ TRUE→isort(isort(x))=isort(x) ⇒ isort(isort(x))=isort(x).
Although the principle of this procedure is rather obvious, the detection of the need for generalization-lemmata and their creation is so far not satisfactorily amenable to automatization, i.e. good heuristics for this purpose are still to be found. (A possible approach to these problems might be achieved by analyzing the way the respective top-level function-symbol of the common subterm under investigation (isort in our example) is defined by the axioms. The results of this analysis might be useful for detecting the need for and the creation of proper generalization-lemmata.)

We shall now introduce two further properties of common subterms that are closely related to over-generalization. They will both provide us with (heuristical) criteria signalizing potentially hazardous constellations.
The first property is the dependency of common subterms which is also discussed in [Hu87]. We shall call analogously a common subterm t *dependent*, if a proper subterm of it occurs somewhere else in the formula where it is not a subterm of an occurence of t.

*Example 3.7*:
In the formula P(g(x),x)∨Q(g(x)), g(x) is a dependent (common) subterm, because x is a proper subterm of g(x) and it occurs in P(g(x),x) at a position where it is not a subterm of a further occurence of g(x).

The criterion of dependency must be regarded as a clue for increased probability of over-generalization. That means, whenever we are dealing with dependent common subterms, we must be aware of the inherent dangers, but we must also keep in mind that generalizing dependent common subterms not necessarily results in over-generalized formulas.

*Example 3.8*:
Let the set of axioms be
x+0=x, x+s(y)=s(x+y),
x•0=0, x•s(y)=x+x•y,
fak(0)=s(0), fak(s(x))=fak(x)•s(x),
F(0,y)=y, F(s(x),y)=F(x,s(x)•y).
When attempting to prove the theorem F(x,s(0))=fak(x), we obtain in the induction-step F(s(t),s(0))=fak(s(t)) what can be rewritten giving F(t,s(t))=fak(t)•s(t). Thus we have the lemma-candidate F(x,s(x))=fak(x)•s(x). The common subterm s(x) is dependent; nevertheless its generalization provides us with the very helpful (and correct) lemma F(x,y)=fak(x)•y.

In those cases where generalizing common subterms does produce over-generalized formulas, the failure mostly is a consequence of a connection that existed between the dependent common subterm and its subterm that caused the dependency. When generalizing we destroy that possibly crucial connection and thus quite often generate useless (wrong) formulas.

*Example 3.9*:
The set of axioms be
app(nil,x)=x
app(cons(y,x),z)=cons(y,app(x,z))
rev(nil)=nil
rev(cons(y,x))=app(rev(x),cons(y,nil)).
The proof of the theorem rev(app(l,l))=app(rev(l),rev(l)) generates in the induction-step rev(app(cons($t_1$,$t_2$),cons($t_1$,$t_2$))) = app(rev(cons($t_1$,$t_2$)),rev(cons($t_1$,$t_2$))). At this point, the common subterm cons($t_1$,$t_2$) is not dependent. After applying the axioms, the equation can be rewritten to app(rev(app($t_2$,cons($t_1$,$t_2$))),cons($t_1$,nil)) = app(app(rev($t_2$),cons($t_1$,nil)),app(rev($t_2$),cons($t_1$,nil))). This way all occurences of cons($t_1$,$t_2$) except one disappeared, thereby acting as ancestors for the common subterm cons($t_1$,nil) in the rewritten equation. cons($t_1$,nil) is a dependent common subterm, because $t_1$ occurs in cons($t_1$,$t_2$) which could not be rewritten. By generalizing cons($t_1$,nil) we destroy the relation that existed and thus finally generate the wrong equation app(rev(app(z,cons(y,z))),x)=app(app(rev(z),x),app(rev(z),x)).

This effect is enhanced if common subterms not only are dependent, but also are “strongly homeomorphically embedded” which is a stronger and therefore more restrictive property. The notion “(strong) homeomorphic embedding” has so far not been used in this context. For this reason, we shall give a definition of what it means if a common subterm is (strongly) homeomorphically embedded.

Definition:
A term s is called *homeomorphically embedded* in a term t, if s$\angle$t, where $\angle$ is a recursively defined relation on terms:
s≡f($s_1$,..,$s_n$)$\angle$g($t_1$,..,$t_m$)≡t iff
(1) f≡g and $s_i\angle t_i$ for all i=1..n=m or

(2) $s \angle t_j$ for one $t_j \in \{t_1,..,t_m\}$
Let $\xi$ be a new symbol, $\varphi[t,s]$ be the term we obtain by replacing all occurences of s in t by $\xi$.
Then s is called *strongly homeomorphically embedded* iff $s \angle \varphi[t,s]$.

*Example 3.10*:
(a) $f(x) \angle f(g(a,x))$
(b) In $P(s(g(s(x))),s(s(x)))$, the subterm s(s(x)) is strongly homeomorphically embedded since $s(s(x)) \angle \varphi[P(s(g(s(x))),s(s(x))),s(s(x))] = P(s(g(s(x))),\xi)$.

Since every subterm that is strongly homeomorphically embedded is also dependent, the criterion of strong homeomorphic embedding provides us with the possibility of a more sophisticated estimation of impending over-generalization beyond the scope of the dependency criterion.
We already mentioned that there can sometimes exist more than one common subterm so that we have to decide which of them to generalize. First of all we must state that every subterm of a common subterm is also a common subterm, provided that it is neither a variable nor a constant. Hence we must decide whether we shall generalize the “bigger” common subterm or the “smaller” ones it contains. For this decision we can make use of the criteria of dependency and strong homeomorphic embedding we introduced above: if we assume a hierarchy “not dependent < dependent < strongly homeomorphically embedded” for common subterms expressing the potential danger of their causing over-generalization, where “<“ means “over-generalization less probable”, we can classify all common subterms utilizing this hierarchy. It is then a good policy to restrict generalization to those commen subterms that are in a minimal class w.r.t <. This strategy can of course be employed the same way if there are several common subterms not necessarily related by the subterm property.

*Example 3.11*:
(a) In $F=P(g(f(a,x),b),b) \vee Q(g(f(a,x),b))$ we find two common subterms g(f(a,x),b) and f(a,x), a subterm of the former. Since g(f(a,x),b) is a dependent common subterm, F is generalized to $P(g(z,b),b) \vee Q(g(z,b))$.
(b) In $P(g(f(a,x),a),b) \vee Q(g(f(a,x),a))$ there are again two common subterms g(f(a,x),a) and f(a,x), which is a subterm of the former. This time, f(a,x) is dependent, whereas g(f(a,x),a) is not dependent. Therefore we generalize the “bigger” common subterm and we obtain $P(z,b) \vee Q(z)$.
(c) In $P(f(z),s(x),s(y)) \vee Q(z,s(x),s(y)) \vee R(f(z))$ we have three common subterms f(z), s(x) and s(y). Since f(z) is dependent, it is suggested to generalize s(x) and s(y) only, thus generating $P(f(z),u,v) \vee Q(z,u,v) \vee R(f(z))$.

Another preference strategy concerning generalization of common subterms is based on the so-called “common subterms with primary occurences” ([Au77]). In this case we confine the generalization to those common subterms that are located on potential induction places. Since generalization replaces them by (new) variables, we should have a good chance of finding a suitable induction variable in a later proof. All other common subterms are left unchanged.

*Example 3.12*:
The set of axioms be {x+0=x, x+s(y)=s(x+y)}.
Suppose we want to generalize the equation s(x)+(s(y)+s(z))=(s(x)+s(y))+s(z). Then it suffices - according to what we just argued- to generalize s(y) and s(z), since s(x) does not occur on a potential induction place. And indeed, the "weaker" generalization s(x)+(u+v)=(s(x)+u)+v allows as easy a proof as the "complete" generalization w+(u+v)=(w+u)+v. So, this strategy enables us to select a satisfactory subset of the common subterms that are to be generalized, and it thereby supports us in our struggle to avoid over-generalization.

Under certain conditions it is recommendable to generalize common subterms although they are not located at potential induction places and even though they are dependent. This is the case if the common subterms have a variable as one of their subterms that appears suitable for the use as induction variable (i.e. it occurs on a potential induction place somewhere else in the formula). By generalizing them we achieve a disconnection from the future induction variable, what makes the proof easier in some cases.

*Example 3.13*:
The set of axioms be the same as in example 3.12.
Although the common subterm s(x) in the equation s(x)+(y+x)=(s(x)+y)+x is not positioned at a potential induction place, its generalization impressively improves provability:
In the induction-step we obtain:
hypothesis: s(t)+(y+t)=(s(t)+y)+t
induction-step: s(s(t))+(y+s(t))=(s(s(t))+y)+s(t) which can be rewritten giving s(s(s(t))+(y+t)) = s((s(s(t))+y)+t). The proof is stuck at this point unless we add a further lemma (s(x)+y=s(x+y)). If we use the generalized version z+(y+x)=(z+y)+x there are no problems (provided that we use x as induction variable; see also section 2.1).

To conclude this overview on aspects of generalizing common subterms we shall classify the heuristic realizing this type of generalization according to the criterion of sufficiency introduced at the beginning of this section. We already argued that every lemma-candidate L is sufficient. Since for every formula L' generated from L by generalization (of common subterms) L'→L holds, L' is also sufficient. This follows from the monotony of deduction.

3.3.2. Creating common subterms with the help of the induction hypotheses
In the preceding section 3.3.1 we have experienced a heuristic for lemma-generation which is without doubt considerably powerful and thus very important. The crucial point for its applicability is the existence of common subterms. These can - if not yet present - be created under certain conditions. One of two methods with that aim is the subject of this section. The second one will be discussed in section 3.3.3.
The method we want to introduce depends on the availability of induction hypotheses in form of pure equalities. (More complex hypotheses must be declined because they would complicate the procedure to a degree not acceptable for practical application.) The basic idea consists in rewriting formulas (lemma-candidates) we would like to contain common subterms with the hypotheses so

that common subterms are generated.

*Example 3.14*:
Suppose we have the following set of axioms
rev(nil)=nil
rev(cons(x,y))=app(rev(y),cons(x,nil))
app(nil,x)=x
app(cons(x,y),z)=cons(x,app(y,z))
and we want to prove the theorem rev(rev(x))=x.
While the base case is rather trivial, we obtain in the induction-step rev(rev(cons(t',t)))=cons(t',t) which can be rewritten to rev(app(rev(t),cons(t',nil)))=cons(t',t). So far there are no common subterms. If we use the hypothesis rev(rev(t))=t from right to left in the right-hand side of the equation generated in the induction-step, we obtain rev(app(rev(t),cons(t',nil))) = cons(t',rev(rev(t))). Hence we created the common subterm rev(t) and we finally can generate the lemma rev(app(x,cons(y,nil)))=cons(y,rev(x)).

The use of the method just outlined does of course not change the property of every lemma-candidate to be sufficient. Therefore a generalization of common subterms combined with the creation of common subterms with the help of the induction hypotheses (equalities) always produces sufficient lemmata.
(Remark: Since we need hypotheses, this method naturally can only be applied in the induction-step.)

3.3.3. Weakening with hypotheses
We shall now present a second strategy with the aim of creating common subterms so as to be able to utilize the generalization of common subterms. As in section 3.3.2 we shall again make use of the hypotheses for this purpose. The considerations forming the foundations of this heuristic are described in the sequel.
When proving an inductive theorem, we dispose in the induction-step of a set of hypotheses $\{hyp_1,..,hyp_n\}$ and the conclusion C (all of them are instances of the theorem to be proved). We then have to show that the implication $\forall x_1,..,x_m:[hyp_1\wedge..\wedge hyp_n]\rightarrow\forall x_1,..,x_m:[C]$ holds, where $x_1,..,x_m$ are the variables occuring in the hypotheses as well as in the conclusion. During the proof we infer conclusion-formulas, i.e. derivates of the conclusion C, so that it makes sense to regard a formula $\forall x_1,..,x_m:[hyp_1\wedge..\wedge hyp_n]\rightarrow\forall x_1',..,x_k':[CF]$ as a derivation of the initial implication, where CF represents a conclusion-formula with variables $x_1',..,x_k'$, which are not necessarily the same variables as in the hypotheses. Our hope is that this implication contains common subterms which CF alone does not contain. (In other words, we treat the whole implication as raw material for lemma-generation instead of using CF only.) In order to improve our chances of finding common subterms in the implication at hand, we transform it into $F\equiv\forall x_1,..,x_m,x_1',..,x_k':[hyp_1\wedge..\wedge hyp_n\rightarrow CF]$, which is of course not an equivalent formula. If we can find common subterms in F suitable for being generalized, then we accomplished what we intended. (Otherwise the heuristic failed.) It is possible that in the case of a success of the heuristic not all of the $hyp_i$ contributed to the creation of common subterms. Since every $hyp_i$ in the conjunction $hyp_1\wedge..\wedge hyp_n$ weakens the resulting

formula (which is supposed to be employed as a lemma later on), what is not at all a desirable effect in inductive proofs (we already discussed this issue), we discard all $hyp_i$ from the antecedent that did not help in creating the common subterms finally generalized. This way we can minimize the negative effect of weakening while preserving its intended purpose.

*Example 3.15*:
We give a subset of the axioms defining insertion-sort and the ORD-predicate (see [Gr90] and appendix A for details) which will suffice for demonstration.
isort(nil)=nil
isort(cons(x,nil))=cons(x,nil)
isort(cons(x,cons(y,nil)))=ins(x,isort(y))
If we want to prove ORD(isort(x)), we shall come up with ORD(isort(cons(t',t))) in the induction-step which can be rewritten giving ORD(ins(t',isort(t))). According to the method just described we generate (ORD(isort(x))∧ORD(isort(y)))→ORD(ins(x,isort(y))) (recall that the complete implication acts as lemma-candidate), since ORD(ins(x,isort(y))) alone does not host common subterms. We can now generalize the independent common subterm isort(y) and we obtain (ORD(isort(x))∧ORD(z))→ORD(ins(x,z)). ORD(isort(x)) is then erased from the antecedent conjunction since it did not contribute to the generation of the common subterm isort(y). Thus we finally have the lemma ORD(z)→ORD(ins(x,z)).

The heuristic just presented is sufficient, because ∀x:[P(x)→Q(x)] implies ∀x:[P(x)]→∀x:[Q(x)]. The rest follows from the fact that every conclusion-formula is sufficient and that the hypotheses together with the axioms belong to the presuppositions.

3.3.4. Minimal mismatching subterms
The key to success in the induction-step of inductive proofs lies in the ability to utilize the hypotheses. Sometimes, an inference involving a hypothesis (and preferably a conclusion-formula) is only prevented by a small number of subterms in both hypothesis and the second formula used in the inference.

*Example 3.16*:
(see also the isort-example in [Gr90] and appendix A for details on the axiomatization)
theorem: perm(x,isort(x)), i.e. the (ordered) list computed by isort is a permutation of its input-list;
From the conclusion perm(cons(t',t),isort(cons(t',t))) we can derive the conclusion-formula el(t',ins(t',isort(t)))∧perm(t,del(t',ins(t',isort(t)))). Here, the subterm isort(t) in the hypothesis perm(t,isort(t)) and the subterm del(t',ins(t',isort(t))) in the conclusion-formula prevent the desired inference.

An approach to solve this problem caused by not unifiable (mismatching) subterms is based on the introduction of equations (as lemmata) that resolve the mismatch. Thus, in the example above, we could add the equation isort(x)=del(y,ins(y,isort(x))) which might even be simplified by generalizing the common subterm isort(x) giving z=del(y,ins(y,z)). This method that reflects the principles of "minimal mismatching subterms" ([Hu87]) depends on the extraction of those

subterms that are the main reason for a mismatch (non-unifiability). These subterms are denoted "minimal mismatching subterms" and are defined as follows.

Definition:
Let $F_1 \equiv P(s_1,..,s_n)$, $F_2 \equiv P(t_1,..,t_n)$ be two literals with no variables in common, $p \in (O(F_1) \cap O(F_2))-\{\varepsilon\}$, $s \equiv \theta(F_1|p)$, $t \equiv \theta(F_2|p)$, where $\theta$ is a substitution with the property that whenever $q \in (O(F_1) \cap O(F_2))-\{\varepsilon\}$ and $F_i|q \equiv x \in V$ ($i \in \{1,2\}$), then there is $q' \in (O(F_1) \cap O(F_2))-\{\varepsilon\}$ (possibly $q=q'$) so that $F_i|q' \equiv x$ and $\theta(x)=\theta(F_{3-i}|q')$ or $F_{3-i}|q \in V$ and $\theta(F_{3-i}|q) \equiv x$ for all such q. ($\theta$ can be understood as "that substitution which brings $F_1$ and $F_2$ as close to successful unification as possible".)
s and t are called *mismatching subterms*, if there is no substitution $\sigma$ so that $\sigma(s)=\sigma(t)$ holds.
s and t are called *minimal mismatching subterms* (resp. [s,t] is called a pair of minimal mismatching subterms), if s and t are mismatching subterms and if $s \equiv f(s_1',..,s_m')$, $t \equiv g(t_1',..,t_k')$, then $f \neq g$.

*Example 3.17*:
$F_1=P(f(g(x)),b)$, $F_2=P(f(h(a,x)),b)$;
Both [f(g(x)),f(h(a,x))] and [g(x),h(a,x)] are pairs of mismatching subterms. But only the latter is a pair of minimal mismatching subterms.

Note:
An obvious procedure that computes all pairs of minimal mismatching subterms when given two literals $F_1$ and $F_2$ as input, where $F_1$ and $F_2$ match the requirements of the definition above, is very close to Robinson's unification algorithm ([Ro71]). The difference is that, whenever we run into term-pairs that would cause Robinson's algorithm to exit with failure (i.e. there is no unifier), these term-pairs are collected in a set $TP_E$ that will act as output when the inference process terminates, containing all pairs of minimal mismatching subterms.

We shall now focus our attention on the question how we should utilize the equations associated with the set of term-pairs contained in $TP_E$.
Basically there are two alternatives, assuming $TP_E$ is not empty. (Otherwise the two investigated literals $F_1$ and $F_2$ are unifiable, thus making an application of this strategy pointless.)
(A) We can use the equations generable from the term-pairs in $TP_E$ all or in part as lemmata (possibly after performing some form of generalization).
(B) We can create a conjunction of all equations or a part thereof and generate a lemma by replacing the literal belonging to the conclusion-formula (let us say $F_2$), which was used to compute $TP_E$, by this conjunction. An additional generalization step is recommendable. (Recall that the idea is to take $F_1$ from the hypothesis and $F_2$ from a conclusion-formula, since the reason for not being able to perform a successful inference between the two of them might reside in these two literals.)
Thus for the isort-example given above we obtain for instance:
(A) x=del(y,ins(y,x)) as lemma (which is not sufficient)
(B) el(y,ins(y,x))$\wedge$x=del(y,ins(y,x)) (which is sufficient)

In general, neither alternative (A) nor (B) can be sufficient since we consider only one literal ($F_1$) of one hypothesis per application of this heuristic. Hence in the case where the hypotheses are more complex than just being made of one literal sufficiency cannot be assured.
(Remark: It is of course allowed that $F_1$ or/and $F_2$ are negated.)

3.3.5. Rippling out

The so-called rippling-out strategy (see [Bu88] for basic version, [BMSI90] for extensions) is dealing with the descrepancy existing between the conclusion in the induction-step and the hypotheses due to the introduction of a constructor-function-symbol. “Rippling out” is used to deminish the descrepancy so that the hypotheses finally become applicable. This can (in the basic version [Bu88]) be achieved by moving (“rippling”) the disturbing constructor-function-symbols to the outside (i.e. towards top-level) of the formula representing the conclusion through applications of suitable equations.

*Example 3.18*:
Let the set of axioms be {x+0=x, x+s(y)=s(x+y)}.
If we want to prove the inductive theorem x+y=y+x (inducing on y), we obtain in the induction-step x+s(t)=s(t)+x yielding s(x+t)=s(t)+x after rippling out the constructor s on the left-hand side of the equation. A fruitful application of the hypothesis x+t=t+x is prevented by the constructor s on the right-hand side of the equation s(x+t)=s(t)+x. If we could also ripple out constructor s on the right side (using the equation s(x)+y=s(x+y) for instance), we should clear the way for success.

We shall now give a general definition of “rippling out”.

Definition:
Let c be a n-ary constructor, n>0, f a m-ary function symbol, m>0, f no constructor.
A transformation of a (sub-)term $t \equiv f(t_1,..,t_{i-1},c(s_1,..,s_{j-1},s_j, s_{j+1},..,s_n),t_{i+1},..,t_m)$ into $t' \equiv c(s_1,..,s_{j-1}, f(t_1,..,t_{i-1},s_j, t_{i+1},..,t_m),s_{j+1},..,s_n)$ is called *rippling out* (constructor c).

While this technique was intended to guide rewrite-steps, i.e. the equations are already at hand, their application only needs to be forced, we shall not contemplate this issue here. Instead, we shall make use of rippling out for generating the necessary equations as lemmata, thus constructing a further method for automatic lemma-generation.
[Bu88] and [BMSI90] also describe some other forms of rippling such as a transformation of $f(t_1,..,t_{i-1},c(s_1,..,s_j,..,s_n),t_{i+1},..,t_m)$ into $f(t_1,..,t_{i-1},s_j,t_{i+1},..,t_m)$, where the constructor c disappears. Since we want to use this method for lemma-generation, we shall content ourselves with the version presented in the definition above. So we can see here a potential for further investigations and extensions of lemma-generation based on rippling techniques.
But for a start, it is satisfactory to concentrate on the creation of (rippling) equations t=t’ via rippling out, proceeding from a term t as it is outlined in the definition. According to the principle of rippling, it is sufficient to generate such an equation t=t’ only if the constructor c involved in the rippling process was introduced when creating an instance of the theorem due to the induction-step.

At first sight, creating t’ and thereby the equation t=t’ seems to be a simple task. But if we take

a closer look, we find out that the index j, which plays an important role in the whole process, is in general not clearly defined and thus can denote any arbitrary argument of c. (In the example above there was no problem because $c \equiv s$ was 1-ary and therefore $j=1$ was the only possible choice.) In our struggle to uphold provability of the equation to be generated, we cannot accept an arbitrary choice of j, because the selection of j will in general have a decisive influence on it. Hence we have to use at least heuristical methods with the aim of choosing j properly. (It is of course possible to select several distinct indices so that several equations can be generated.)
A first approach is the use of sorts if available. It is clear that the sort of $s_j$ must be the same as the sort of $c(s_1,..,s_n)$, or a subsort thereof. But sort-information can only be restrictive and will in general not determine the choice of j. Therefore, and of course for the case where no sort-information is available, we have to think of other methods.
A quite obvious method for deciding which j to take stems directly from the overall purpose of rippling out, i.e. attempting to make the hypotheses fruitfully applicable. It is therefore straight forward to select j so that the related equation t=t' gives rise to the intended effect. But deciding whether such an equation t=t' has the desired effect or not can sometimes be a computationally exhaustive task.

*Example 3.19*:
Let the axioms be
$x+0=x$, $x+s(y)=s(x+y)$,
$even(0)$, $\neg even(s(0))$, $even(x) \leftrightarrow even(s(s(x)))$;
The proof of $even(x+x)$ yields in the induction-step $even(s(t)+s(t))$ which can be rewritten to $even(s(s(t)+t))$. The suitability of the equation $s(x)+x=s(x+x)$ resp. $s(x)+y=s(x+y)$ can only be discovered after an additional inference involving $even(x) \leftrightarrow even(s(s(x)))$, since only then it becomes visible that the hypothesis $even(t+t)$ can be applied.

This problem can even be more complicated.
On the one hand, the induction variable can occur at several positions, a circumstance which causes the constructor introduced in the induction-step to appear on several locations. Thus several different equations might be necessary to ripple all of them out, what truly can make the decision, whether a rippling equation has the desired effect or not, rather difficult.
On the other hand, the induction variable can be very "deep" in the term structure. If this happens, we have to ripple out cascade-like, using a set of possibly different rippling equations.
Both cases, which can of course occur in combinations, do not exclude the need of further inferences before, during or after the application of rippling equations. So, proceeding in this manner brings us close to proof planning, an interesting but also very extensive field of research beyond the goals of this report. Therefore we want to present a simpler, heuristical method for finding an appropriate index j (or several indices if possible).
The idea is to investigate the equational axioms s=t in order to detect constellations where $s \equiv f(t_1,..,t_{k-1},c(s_1,..,s_{l-1},s_l,s_{l+1},..,s_n),t_{k+1},..,t_m)$ and $t \equiv c(s_1,..,s_{l-1},f(t_1,..,t_{k-1},s_l,t_{k+1},..,t_m),s_{l+1},..,s_n)$ or where s and t play contrary roles.

Note: The index k is assumed to be different from i; otherwise the creation of a rippling equation

would be pointless because it already exists as an axiom.
All indices l found this way are collected in the set $\Gamma$. If $\Gamma$ is empty, then this is a strong clue that rippling out might be illegal (i.e. results in incorrectness). It is suggested to avoid the generation and application of rippling equations under these conditions. If $\Gamma$ is not empty, then for every l in $\Gamma$ we generate the respective rippling equation for j=l, and we may then use them as lemmata. Note that this proceeding does not at all guarantee correctness of the generated lemmata (rippling equations), and it of course does not necessarily allow fruitful applications of hypotheses. Nevertheless, the heuristic has turned out to be reliable and effective in practical use.
A further improvement can be achieved if we replace all variables on potential induction places (and only those variables) by new ones. Thus we can again accomplish a disconnection of future induction variables from the rest with all the already discussed advantages and disadvantages.

This heuristic for lemma-generation is not sufficient. For the counterexample we start from the axioms of example 3.19. If we prove the theorem $(even(x)\wedge even(y))\rightarrow even(x+y)$ and select x as induction variable (what is a not a good choice), we obtain $(even(s(t))\wedge even(y))\rightarrow even(s(t)+y)$ in the induction-step. Adding the lemma $s(x)+y=s(x+y)$ does not conclude the proof. But in combination with a heuristic that will be described in the following section, the proof is finally found despite the poor choice of the induction variable.

Remark:
Based on the idea of using the axioms as a source of knowledge leading to appropriate rippling equations, more sophisticated methods can be elaborated. But as such a proceeding also tends towards proof planning, we did not consider it here, thus leaving plenty of room for further investigations.

### 3.3.6. Adding conjunctions

We already know from earlier discussions that stronger theorems often allow an easier proof by induction because of improved usefulness of the hypotheses. One way to strengthen formulas has been introduced when presenting generalization (of common subterms). In this section we shall describe another method aiming at strengthening formulas. Its principle (see also [Hu87]) is based on adding a conjunction B ($B=B_1\wedge..\wedge B_n$, $n\geq 1$) to a formula A so that we obtain the stronger formula $A\wedge B$. The problem arising here lies in the lack of knowledge about B. While any formula $B\neq TRUE$ meets all basic requirements, we also have to keep in mind that the assumed provability of A is to be preserved. This means that we have to be careful about the choice of B.
There are certainly a lot of methods conceivable which all operate on the principle of adding conjunctions. Due to the general character of this report, only one heuristic will be described. It does not operate on lemma-candidates, but it depends on an induction-step being proved, as we shall see. (For this reason, we could call it a “constructive” heuristic as opposed to the usual case of a “manipulating” one.)
As we already stated, the main problem consists in ascertaining the provability of the generated formula A’. Therefore we should use for the conjunction only formulas which most probably can be proved by induction. For this purpose a conjunction built of hypotheses and instance of an induction-step with the ground-term symbols $t_i$ replaced by new variables seems perfectly

suitable. We thus obtain $A'=A[x\leftarrow x_1]\wedge..\wedge A[x\leftarrow x_n]\wedge A[x\leftarrow f(x_1,..,x_n)]$, where x is the induction variable, f the constructor-function of the respective induction-step. (This implies that the heuristic is applied during the induction-step involving f.) If A' is used as a lemma, it obviously concludes at once the proof (of the respective induction-step) since $A'\rightarrow A$. Therefore the heuristic is sufficient. We conclude this section with an example.

*Example 3.20*:
Let the axioms be
$x+0=x$, $x+s(y)=s(x+y)$,
$even(0)$, $\neg even(s(0))$, $even(x)\leftrightarrow even(s(s(x)))$,
$\neg odd(0)$, $odd(s(0))$, $odd(x)\leftrightarrow odd(s(s(x)))$;
When proving $even(x)\vee odd(x)$ we obtain $even(s(t))\vee odd(s(t))$ in the induction-step. There is no possibility of ever applying the hypothesis $even(t)\vee odd(t)$. The only axioms we could use would cause inflation. A generalization of common subterms might produce the original conjecture. With the presented heuristic we create the formula $(even(x)\vee odd(x))\wedge(even(s(x))\vee odd(s(x)))$, which is provable without problems, because the hypothesis resolves the induction-step crosswise (after application of the above equivalences).
(Remark: The success of the heuristic in this example is surely favoured by the "double-step" definition of even and odd. But as already stated, this is just one simple method in the vast field of methods based on adding conjunctions, pointing out inherent potential.)

### 3.4. Application hierarchy of the heuristics for lemma-generation

Meanwhile we know a number of heuristics for lemma-generation and we have to ask ourselves how this complies with our postulated goal of creating exactly one lemma. For usually more than one heuristic will be applicable so that solely due to the multitude of heuristics at hand several lemmata are creatable from one lemma-candidate, not to speak of the vast number of lemmata that might be generated on account of a large amount of conclusion-formulas (Recall that every conclusion-formula is utilized as a lemma-candidate which (in part) are used by the heuristics for lemma-generation.)
One way to cope with the problem of generating more than one lemma from only one lemma-candidate is to develop an application hierarchy of the lemma-generating heuristics which reflects the reliability and anticipated performance (based on experimental results) of each individual heuristic. The idea is to start with the heuristic that is considered best and to continue down the hierarchy as long as no heuristic produces a lemma. As soon as a lemma could be created, the process stops. If none of the heuristics was able to deliver a lemma, then we have to question the quality of the respective lemma-candidate, or we use the lemma-candidate itself as a lemma without further modification.
This concept can of course be refined and extended. So it is possible - as we have seen in example 3.20 - that for instance the generalization of common subterms might produce a lemma which is equivalent to the original conjecture. Then it is recommendable to have at hand a mechanism for detecting such events and for reacting in a proper way, e.g. by trying to apply other heuristics.

## 4. Conclusion and outlook

In the past sections, we have come to know a method enabling us to extend any theorem prover for PL1EQ (first order predicate logic with equality), so that it can perform inductive proofs. The first step towards this goal, using the principles of structural induction, was not very hard, but yet was rather user-dependent, i.e. it did not have the degree of automatization we should like to achieve. Therefore a range of features such as automatic choice of the induction variable, detecting the need for (further) inductive lemmata etc. and above all automatic generation of inductive lemmata were added, all of them with heuristical foundations. Hereby the generation of inductive lemmata received highest attention due to its crucial importance for successful proofs. The way we realized this is characterized by a two-step procedure that is activated during a proof attempt. The formulas available at that time are investigated in the first step, extracting all those formulas that were inferred with participation of the conclusion (goal) or a derivate thereof. These so-called conclusion-formulas form - after slight modifications - the set of so-called lemma-candidates. This set is reduced (using some criteria also discussed) in order to get closer to the ideal goal of generating exactly one (correct and useful) lemma. The second step applies a number of heuristics (in a certain order defined by the "hierarchy of application") to the remaining lemma-candidates, thus finally creating lemmata. Because of the heuristical concept of the whole proceeding, it cannot be guaranteed that the created lemmata belong to the inductive theory we are currently interested in. But experiments and practical experience have shown an encouraging behaviour.

With all these tools at hand, we are of course still far away from a really efficient and acceptably automatized inductive theorem prover. The heuristics for lemma-generation and also the heuristics which are used to reduce the set of lemma-candidates need furter elaboration. Moreover, more "intelligent" methods are required to solve more challenging problems (for instance problems associated with the binomial coefficient).

A severe problem we did not touch throughout this report are the difficulties we run into when dealing with existential quantification. The solution to this problem lies in program synthesis, which is a complex domain that is clearly beyond the scope of this introductory presentation of how to handle the basic issues of inductive proving. The reader be referred to [Bi91] and related literature.

**Acknowledgment**

I thank Jörg Denzinger for helpful comments on contents and outfit of this report and for encouraging me to write it in the first place.

**Appendix A**

The following example is intended to demonstrate the capabilities of an inductive prover equipped with the basic mechanism structural induction and additional features introduced in this report, including automatic lemma-generation.
In this example we want to prove the correctness of a sorting algorithm, namely insertion-sorting. The associated function isort operates on lists of natural numbers, i.e. isort(x)=y, where x is an arbitrary list of natural numbers, and y is a list with the elements of x in ascending order. The proof of correctness is twofold:
(1) We must prove that isort always produces lists whose elements are in ascending order (w.r.t. the usual <-relation on natural numbers). This will be expressed by ORD(isort(x)) (for all x).
(2) We also have to show that no element of the input-list disappeared and no elements were added, i.e. the output is a permutation of the input, denoted by PERM(x,isort(x)) (for all x).
All the axioms necessary for specifying isort, ORD, PERM and related predicates and functions are listed below (see also [Gr90]):

(01) $0 \leq x$
(02) $\neg(s(x) \leq 0)$
(03) $s(x) \leq s(y) \leftrightarrow x \leq y$
(04) isort(nil)=nil
(05) isort(cons(x,y))=ins(x,isort(y))
(06) ins(x,nil)=cons(x,nil)
(07) $x \leq y \rightarrow$ins(x,cons(y,z))=cons(x,cons(y,z))
(08) $\neg(x \leq y) \rightarrow$ins(x,cons(y,z))=cons(y,ins(x,z))
(09) ORD(nil)
(10) ORD(cons(x,nil))
(11) ORD(cons(x,cons(y,z))) $\leftrightarrow (x \leq y \wedge$ORD(cons(y,z)))
(12) PERM(nil,nil)
(13) $\neg$PERM(nil,cons(x,y))
(14) PERM(cons(x,y),z) $\leftrightarrow$(EL(x,z) $\wedge$PERM(y,del(x,z)))
(15) $\neg$EL(x,nil)
(16) EL(x,cons(y,z)) $\leftrightarrow (x=y \vee$EL(x,z))
(17) del(x,nil)=nil
(18) $x=y \rightarrow$del(x,cons(y,z))=z
(19) $\neg(x=y) \rightarrow$del(x,cons(y,z))=cons(y,del(x,z))

Let us begin the proof for ORD(isort(x)) (remember that we use implicit $\forall$-quantification). The variable x is the only possible induction variable. (Note: All induction variables we shall use for the proofs are identical to those that the heuristic for selecting the induction variable would propose. See section 2.1.)
Furthermore we have $FS_C$={cons}, $C_C$={nil}.

base-case: ORD(isort(nil)) $\Rightarrow$ ORD(nil).

hypothesis: ORD(isort(t)); since cons is binary, we should have two hypotheses; but one of these can be omitted assuming knowledge about sorts as we shall do throughout the rest of the example.
induction-step: ORD(isort(cons(t',t))) ⇒ ORD(ins(t',isort(t)));
We have at this point no chance to continue our work on the conclusion. This fact should encourage the system to stop the proof here and to attempt to generate a lemma. As we have already seen in the course of the description of the heuristic called "weakening with hypotheses", an application of that method provides us with the sufficient lemma ORD(x)→ORD(ins(y,x)). Therefore it suffices to concentrate on its proof:

base-case: ORD(nil)→ORD(ins(y,nil)) ⇒ ORD(cons(y,nil)).
hypothesis: ORD(t)→ORD(ins(y,t))
induction-step: ORD(cons(t',t))→ORD(ins(y,cons(t',t)));
We have two alternatives (cases):
Applying axiom (07) leads us to (y≤t'∧ORD(cons(t',t)))→ORD(cons(y,cons(t',t))) which is trivialized when using axiom (11). This does of course not complete the proof, since we virtually splitted the proof into the two cases where y≤t' and the following one where ¬(y≤t'.) Hence we have to consider the second alternative utilizing axiom (08) and we obtain (¬(y≤t')∧ORD(cons(t',t)))→ORD(cons(t',ins(y,t))). The only possible inference we can employ in this situation is an application of axiom (03), yielding inflated formulas. We again fall back upon lemma-generation, thus generating the lemma (¬(y≤x)∧ORD(cons(x,z)))→ORD(cons(x,ins(y,z))), which is sufficient. (Note: This lemma is identical to the lemma-candidate, since none of the available heuristics could modify it.)

The proof of this lemma proceeds as follows (using z as induction variable).
base-case:
(¬(y≤x)∧ORD(cons(x,nil)))→ORD(cons(x,ins(y,nil)))⇒¬(y≤x)→ORD(cons(x,cons(y,nil)));
the only meaningful inference involves axiom (11), resulting in ¬(y≤x)→(x≤y∧ORD(cons(y,nil))) ⇒ ¬(y≤x)→x≤y ("totality of ≤"). Now, only the inflating axiom (03) is applicable. Consequently, ¬(y≤x)→x≤y is chosen as (sufficient) lemma. For its proof, we have to alter $FS_C$ and $C_C$ to {s} and {0} respectively. The choice of the induction variable does not matter for this proof; we select x.

base-case: ¬(y≤0)→0≤y ⇒ ¬(y≤0)→TRUE.
hypothesis: ¬(y≤t)→t≤y;
induction-step: ¬(y≤s(t))→s(t)≤y;
We are once more in the unfortunate position where only axiom (03) can be applied. Moreover, the only "uninflated" lemmata generable are ¬(y≤s(z))→s(z)≤x (which is a specalization of the original conclusion) and ¬(y≤x)→x≤y (by generalization of common subterms). The latter is equivalent to the initial lemma which is to be proved. Both possibilities are no help. But the phenomenon of generating the conjecture, which has to be proved, by means of generalization of common subterms recommends the use of induction on several variables. We therefore prove ¬(y≤s(t))→s(t)≤y by a further induction on y, preserving the hypothesis ¬(y≤t)→t≤y.
base-case 2: ¬(0≤s(t))→s(t)≤0 ⇒ FALSE→s(t)≤0.
hypothesis 2: ¬(t'≤s(t))→s(t)≤t'

induction-step: $\neg(s(t')\leq s(t))\rightarrow s(t)\leq s(t')$. Applying axiom (03) this time provides us with $\neg(t'\leq t)\rightarrow t\leq t'$, and the inherited hypothesis concludes the proof (of this induction).

We can now continue the proof of $(\neg(y\leq x)\wedge ORD(cons(x,z)))\rightarrow ORD(cons(x,ins(y,z)))$ with the induction-step. Naturally, we shall make use of the lemma $\neg(y\leq x)\rightarrow x\leq y$, because we already used (and proved) it for the base-case.

hypothesis: $(\neg(y\leq x)\wedge ORD(cons(x,t)))\rightarrow ORD(cons(x,ins(y,t)))$

induction-step: $(\neg(y\leq x)\wedge ORD(cons(x,cons(t',t))))\rightarrow ORD(cons(x,ins(y,cons(t',t))))$. Applying axiom (11) yields $(\neg(y\leq x)\wedge x\leq t'\wedge ORD(cons(t',t)))\rightarrow ORD(cons(x,ins(y,cons(t',t))))$. We have now two complementary alternatives:

(1) Using axiom (07) we obtain $(\neg(y\leq x)\wedge x\leq t'\wedge ORD(cons(t',t)))\rightarrow ORD(cons(x,cons(y,(cons(t',t))))$. Applying axiom (11) twice gives us $(\neg(y\leq x)\wedge x\leq y\wedge y\leq t'\wedge ORD(cons(t',t)))\rightarrow (x\leq y\wedge y\leq t'\wedge ORD(cons(t',t)))$, which simplifies to TRUE after employing the lemma $\neg(y\leq x)\rightarrow x\leq y$.

(2) If we use axiom (08), we are provided with $(\neg(y\leq x)\wedge x\leq y\wedge\neg(y\leq t')\wedge ORD(cons(t',t)))\rightarrow ORD(cons(x,cons(t',ins(y,t))))$. An application of axiom (11) yields $(\neg(y\leq x)\wedge x\leq y\wedge\neg(y\leq t')\wedge ORD(cons(t',t)))\rightarrow(x\leq t'\wedge ORD(cons(t',ins(y,t))))$, which also simplifies to TRUE using the hypothesis.

We have now completed the proof of ORD(isort(x)), and we can turn our attention to the assertion PERM(x,isort(x)):

base-case: $PERM(nil,isort(nil))\Rightarrow PERM(nil,nil)$.

hypothesis: PERM(t,isort(t));

induction-step: $PERM(cons(t',t),isort(cons(t',t)))\Rightarrow PERM(cons(t',t),ins(t',isort(t)))\Rightarrow EL(t',ins(t',isort(t)))\wedge PERM(t,del(t',ins(t',isort(t))))$. We have encountered this constellation when discussing the issues related to “minimal mismatching subterms”. By using the strategy which was described as the second alternative, we obtain the lemma $EL(x,ins(x,y))\wedge del(x,ins(x,y))=y$, which we have to incorporate into the proof since the heuristic for its generation is not sufficient. But this time, it is obvious that it concludes the proof. Hence we have to concentrate on the proof of the lemma. We shall split up the conjunction, what is legal since it is implicitly $\forall$-quantified by convention.

Let us start with the proof of EL(x,ins(x,y)), selecting y as induction variable.

base-case: $EL(x,ins(x,nil))\Rightarrow EL(x,cons(x,nil))$.

hypothesis: EL(x,ins(x,t));

induction-step: EL(x,ins(x,cons(t',t)));

We have to distinguish two cases:

(1) The application of axiom (07) yields $x\leq t'\rightarrow EL(x,cons(x,cons(t',t)))\Rightarrow x\leq t'\rightarrow TRUE$.

(2) Applying axiom (08) gives us $\neg(x\leq t')\rightarrow EL(x,cons(t',ins(x,t)))\Rightarrow\neg(x\leq t')\rightarrow(x=t'\vee EL(x,ins(x,t)))$ with axiom (16). By making use of the hypothesis, we can conclude the proof.

Thus, the only lemma that remains to be proved is $del(x,ins(x,y))=y$. Inducing on y yields:

base-case: $del(x,ins(x,nil))=nil\Rightarrow del(x,cons(x,nil))=nil$;

Two cases arise:

(1) When using axiom (18), we obtain x=x→nil=nil, which is obviously TRUE.
(2) The use of axiom (19) provides us with ¬(x=x)→cons(x,del(x,nil))=nil ⇒ FALSE→ cons(x,nil)=nil and we are done.
hypothesis: del(x,ins(x,t))=t;
induction-step: del(x,ins(x,cons(t',t)))=cons(t',t);
Once again we must distinguish two cases:
(1) Applying axiom (07) generates x≤t'→del(x,cons(x,cons(t',t)))=cons(t',t), which we can transform into TRUE analogously to what we just performed in the base-case.
(2) Using axiom (08) yields ¬(x≤t')→del(x,cons(t',ins(x,t)))=cons(t',t); Two further distinct cases are to be dealt with at this point:
(2a) (¬(x≤t')∧x=t')→ins(x,t)=cons(t',t)
(2b) (¬(x≤t')∧¬(x=t'))→cons(t',del(x,ins(x,t)))=cons(t',t)
The conjecture (2b) can be rewritten to (¬(x≤t')∧¬(x=t'))→cons(t',t)=cons(t',t) by utilizing the hypothesis and we are done.
For a proof of (2a) we apply x=t' and obtain ¬(t'≤t')→ins(t',t)=cons(t',t). The only chance we have to prove this is by making the antecedent false, thus calling for the lemma x≤x. The way we found this lemma seems rather intuitive (if not to say "far fetched") and beyond the limits of the automatic lemma-generation strategies described in this report. But with extensions incorporating heuristics as the one just sketched, the system can be taught to a certainly limited extent to use some kind of "intuition" itself.
Apart from that, trying to prove (2a) by refutation makes the need of x≤x clear without having to rely on more sophisticated lemma-generation heuristics:
Negating and skolemizing (2a), using the skolem-constant a, yields ¬(a≤t')∧a=t'∧¬(ins(a,t)=cons(t',t)). By rewriting with a=t' we obtain (for instance) ¬(a≤a)∧a=t'∧¬(ins(a,t)=cons(a,t)). Thus we have the following lemma-candidate (recall: we must reverse the effects of negation and skolemization) x≤x∨¬(x=y)∨ins(x,z)=cons(x,z), which we utilize as a (sufficient) lemma. The proof thereof imposes no major problems when using x as induction variable, since x≤x alone can be proved by induction without complications (we shall therefore omit the details). Naturally, it would be advantageous to dispose of methods which allow us to recognize the fact that part of the disjunction is not correct w.r.t. to the investigated inductive theory and can therefore be dropped. We did not cover the issues related to this topic. See, for instance, [Pr92].

Finally, we have proved both properties of isort we postulated at the beginning, and thus have confirmed correctness of the related sorting-algorithm.

**Appendix B**

Being able to measure in some way the inflation of formulas or terms is not only advantageous in connection with those methods presented in this report. There are also other fields of application detached from inductive proving, such as detecting divergence in general or utilization as preference strategy (e.g. for selecting the next critical pair in the KB-completion procedure ([KB70])).

But how should we "measure" the inflation of a term (resp. formula)? Expressing the "inflation-degree" of a given term or formula in terms of natural numbers appears to be quite suitable, since this way the well-known ordering < on natural numbers can very conveniently be used to compare two terms or formulas w.r.t. their inflation degree. Thus, for instance, s(x), s(s(x)) and s(s(x))) are assigned the inflation-degrees 0, 1 and 2 respectively. Certainly, we also would like to measure more complicated terms such as f(x,g(f(y,g(f(z,g(a,b)))))).

In order to make clear the principles of the computation of the inflation-degree and in order to ease the understanding of the definitions to come, we shall contemplate the term t≡h(y,f(x,g(f(y,g(f(z,g(f(a,b)))))),c). We can easily discover that t, or to be more exact, its subterm t'=t|2 shows a regular resp. recursive structure we refer to as inflation. This is manifested in the subterm t''=t'|2.1≡f(y,g(f(z,g(f(a,b))))), which shows a similar setup as t' itself. An analogous relation exists between t'' and its subterm t'''=t''|2.1≡f(z,g(f(a,b))). Due to the finiteness of terms it is apparent that this repetition of similar structures will come to an end sooner or later for any term. The important point consists in capturing the number of repetitions so far. In the attempt to solve this task, the places of t' and t'', where the recursive structure terminates, play a substantial role. Here, these places are 2.1.2.1.1 and 2.1.2.1.2:

t'=f(x,g(f(y,g(f(z,g(f(a,b)))))))  t'|2.1.2.1.1=z  t'|2.1.2.1.2=g(f(a,b))
t''=f(y,g(f(z,g(f(a,b)))))  t''|2.1.2.1.1=a  t''|2.1.2.1.2=b

These so-called "critical places" are collected in a set P. Thus, for the example, we have P={2.1.2.1.1,2.1.2.1.2}. The regularity of the structure reveals itself through the occurence of 2.1 (recall that t'|2.1=t'') in the maximal prefix $(2.1)^2$ of all places in P. The exponent (here 2) will later serve to compute the inflation-degree.

For the computation of the inflation-degree of any term t, we have to treat every subterm t' in the outlined manner, i.e. we have to check up all subterms t'|p by computing the associate set $P_p$ and then determine the maximal exponent n such that all $q \in P_p$ can be written as $p^n q'$. We choose the maximum of all these exponents we obtain by considering all subterms t' of t, and we use this value as inflation-degree.

Hence, the inflation-degree of a term $t \equiv h(s^i(x),s^j(x))$ is the maximum of i-1, j-1 and 0 (for the case i=j=0 and consequently t≡h(x,x)).

After the informal presentation of the basic concept, the definitions shall be given.

Definition

Let Var denote the set of variables.

(1) Let t be a term, $p \in O(t)-\{\varepsilon\}$, $q \in O(t) \cap O(t|p)$.

q is a *strong critical place* of t|p and t if q=q'u with

$(t|p)|q' \equiv f(t_1,..,t_n)$, $t|q' \equiv g(s_1,..,s_m)$, $f \neq g$, $n,m \geq 0$ or

(t|p)|q'∈Var, t|q'∉Var or
(t|p)|q'∉Var, t|q'∈Var

(2) Let t, p and q be given as in (1).
q is a *critical place* of t|p and t if
q is a strong critical place or
t|q∈Var, (t|p)|q∈Var, t|q≠(t|p)|q and there is a $q_1$∈O(t|p), $q_1$≠q, (t|p)|$q_1$≡t|q, where $q_1$ is not a strong critical place.
(*Example*:
t≡f(x,f(x,f(y,f(y,g(x)))))
(a) p=2: t|2=f(x,f(y,f(y,g(x))))
Then we have the strong critical places 2.2.2 and 2.2.2.1 (which, according to the definition, are also critical places). q=2.1 is a critical place (not a strong critical place), since t|2.1≡x∈Var, (t|2)|2.1≡y∈Var, y≠x and q=2.1≠1=$q_1$, (t|p)|$q_1$≠t|q, where $q_1$ is not a strong critical place.
(b) p=2.2: t|2.2≡f(y,f(y,g(x)))
In this case, every critical place is also a strong critical place. (They are 2.2 and 2.2.1.)
(c) If t≡f(x,f(y,f(z,g(x)))), then with p=2 we also have only strong critical places (2.2 resp. 2.2.1), because the second occurence of x is at a strong critical place.)

(3) If t is a variable or a constant, then its *inflation-degree* is 0.
Let t be a term, t neither a variable nor a constant. For every p∈O(t) $P_p$:={q | q is a critical place of t|p and (֍)for all u≠ε, uq'=q, u is not a critical place of t|p and t}. For all p∈O(t)-{ε} $m_p$:=min{n | $p^n$q'=q∈$P_p$ and there is no q'' with pq''=q'}. Furthermore, $A_{t'}$:=max{$m_p$ | p∈O(t')-{ε}}.
max{$A_s$ | s=t|u, u∈O(t)} is the inflation-degree of t.

Remark: In definition (3), (֍) is only needed to reduce the cardinality of $P_p$, what is of practical interest, but has no theoretical effect.

Note: $m_p$ is always defined, since $P_p$≠∅ for all p∈O(t)-{ε} for all terms t.

*Examples*:

t≡s(s(s(x))) ⇒ O(t)={ε,1,1.1,1.1.1}
p=1 : $P_1$={1.1} ⇒ $m_1$=2
p=1.1 : $P_{1.1}$={1} ⇒ $m_{1.1}$=0
p=1.1.1 : $P_{1.1.1}$={ε} ⇒ $m_{1.1.1}$=0
Thus $A_{s(s(s(x)))}$=2;
Analogously we obtain $A_{s(s(x))}$=1, $A_{s(x)}$=$A_x$=0. Hence t has inflation-degree 2.

t≡h(s(s(s(x))),s(s(s(s(x))))) ⇒ O(t)={ε,1,2,1.1,1.1.1,1.1.1.1,2.1,2.1.1,2.1.1.1,2.1.1.1.1}
Since $P_u$={ε} for all u∈O(t), we have $A_{h(s(s(s(x))),s(s(s(s(x)))))}$=0.
Furthermore, for the remaining subterms of t we obtain:
$A_{s(s(s(x)))}$=2, $A_{s(s(x))}$=1, $A_{s(x)}$=$A_x$=0, $A_{s(s(s(s(x))))}$=3.

Thus t has inflation-degree 3.

$t \equiv f(x,g(f(y,g(f(a,z,a)),y)),x) \Rightarrow$
$O(t)=\{\varepsilon,1,2,3,2.1,2.1.1,2.1.2,2.1.3,2.1.2.1,2.1.2.1.1,2.1.2.1.2,2.1.2.1.3\}$
$p=1 : P_1=\{\varepsilon\} \Rightarrow m_1=0$
$p=2 : P_2=\{\varepsilon\} \Rightarrow m_2=0$
$p=3 : P_3=\{\varepsilon\} \Rightarrow m_3=0$
$p=2.1 : P_{2.1}=\{2.1.1,2.1.2,2.1.3\} \Rightarrow m_{2.1}=1$
$p=2.1.1 : P_{2.1.1}=\{\varepsilon\} \Rightarrow m_{2.1.1}=0$
...
$p=2.1.2.1 : P_{2.1.2.1}=\{1,2,3\} \Rightarrow m_{2.1.2.1}=0$
...
Hence $A_t=1$. Since $A_s=0$ for all proper subterms s of t, t has inflation-degree 1.

**References**


[Au77] Aubin, R. :
"Mechanizing structural induction"
Parts 1 and 2
Theoretical computer science 9 (1979)
Part 1: pp 329-345
Part 2: pp 347-362

[BHSI90] Bundy, A.; van Harmelen, F.; Smaill, A.; Ireland, A. :
"Extensions to the rippling-out tactic for guiding
inductive proofs."
10th CADE, Kaiserslautern 1990
Springer LNAI 449

[Bi86] Biundo, S.:
"A synthesis system mechanizing proofs by induction"
Proc. 7th ECAI, Brighton 1986

[Bi91] Biundo, S.:
"Automatische Synthese rekursiver Programme als Beweisverfahren"
Informatik-Fachberichte 302
Springer-Verlag, 1991

[BM79] Boyer, R.S.; Moore J.S. :
"A computational logic"
Academic Press, 1979

[Bu88] Bundy, A. :
"The use of explicit plans to guide inductive proofs"
Int. 9th conference on automated deduction
pp 111-120
Springer-Verlag, 1988

[CL73] Chang, C.L.; Lee, R.C. :
"Symbolic logic and mechanical theorem proving"
Academic Press, 1973

[Gr90] Gramlich, B.:
"Completion based inductive theorem proving -
a case study in verifying sorting algorithms"
SEKI-Report SR-90-04

FB Informatik, Universität Kaiserslautern 1990

[Hu87] Hummel, B. :
“An investigation of formula generalization heuristics
for inductive proofs”
Interner Bericht Nr. 6/87 der Universität Karlsruhe, 1987

[KB70] Knuth, D.E.; Bendix, P.B. :
“Simple word problems in universal algebra”
Computational Algebra, J. Leach, Pergamon Press,
1970, pp 263-297

[Ni80] Nilsson, N.J. :
“Principles of artificial intelligence”
SRI International, 1980

[Pr92] Protzen, M.:
“Disproving conjectures”
D. Kapur (ed.)
Proceedings 11th conference on automated deduction
Saratoga Springs, NY (USA)
Lecture notes in computer science (in artificial intelligence) vol. 607, 1992
pp 340-354

[Ro71] Robinson, J.A.:
“Computational logic, the unification computation”
Machine Intelligence, Vol.6, Edinburgh Univ. Press, pp 63-72
Scotland 1971