On Generating of Proofs

Vilém Vychodil
Dept. Computer Science, Palacký University, Tomkova 40, CZ-779 00, Olomouc, Czech Republic
email: vilem.vychodil@upol.cz

Abstract— **The paper introduces an approach to automated deduction. Described is a method for generating of proofs which are considered as finite sequences of formulas. Unlike the most common approaches to automated deduction which are based mainly on the resolution principle, our method resembles principles of generating of programs which are usually used in genetic programming. We present an overview of the method and present two case studies. We argue that even in its preliminary stage, our method can be helpful to experts who need to find proofs or counterexamples.**

Keywords— **automated deduction, proofs, randomness**

I. Introduction

Various methods of automated deduction play a crucial role in computer science. It has long been recognized that formal deduction can be used to verify critical properties of software and hardware components, it can be used to infer statements about a system which is described by formulas, and it is a formal basis of various declarative programming languages. One of the directions of formal deduction in computer science lead to construction of automated theorem provers. An automated theorem prover can be seen as an abstract machine which, given a set of formulas as its input, infers new formulas from the input ones trying to satisfy a given goal (e.g., to show that certain formulas can be inferred). The efforts to create theorem provers have intensified after the inception of the resolution principle which seemed to suit well the idea of automated deduction. The resolution principle, which is due to Robinson, is now most commonly associated with logic programming and the programming language PROLOG [11].

The basic idea of provers based on the resolution principle is to find a contradiction in a so-called resolution closure which indicates that a given formula (usually called a goal) is provable from input formulas. Provers that use the resolution principle rely on a systematic search of state space which may be infinite. This brings into the problem several technical complications (e.g., the prover can loop in an infinite branch of the state space without finding a contradiction, thus "running forever" without giving an answer) as well as some theoretical limitations related to decidability [12]. If an automated prover stops (for a given input), it usually outputs "Yes" (formula representing the goal is provable from the input formulas), possibly with a substitution indicating which instance of the goal is provable, or "No" (not provable). This behavior is typical for the majority of compilers of logic programming languages including PROLOG.

Although the resolution principle is suitable for automated deduction and easy to implement, one may argue that the principle is far away from the human-like deduction because nobody really deduces things by finding contradictions in resolution closures. Instead, one usually starts with initial statements (representing facts) and then draws conclusions (new statements) using (some) inference rules applied on the initial statements and conclusions made in previous steps. This procedure is usually combined with finding counterexamples. By a counterexample is meant a model of the initial statements in which certain statement is not true. Thus, if one uses a sound inference (this is a basic assumption of each "rational inference"), then by a counterexample it can be shown that a statement cannot be proved. During the inference, one is usually guided (and sometimes misguided) by her/his experience. An important role is also played by randomness supported by knowledge of the problem domain: for instance, if there are several ways to proceed which all seem to be reasonable (i.e. they seem to lead to a desirable result), one often randomly tries them one by one. The process of proving statements and disproving statements by counterexamples is repeated until one finally arrives to an interesting conclusion. In our paper we outline a method of automated inference which is not based on the resolution principle but tries to emulate the human-like inference procedure of "finding proofs and counterexamples".

One of the motivations for our method is the practical advantage of generating of proofs as sequences of formulas. Namely, the output of automated provers based on the resolution principle is usually only "Yes" (provable) or "No" (not provable). It is often the case that a user wishes to see a proof, i.e. she/he not only is interested in the fact that a statement is provable but she/he wants to see its proof because the proof is "a record of how the statement was proved" and can contain interesting constructions which may be used further. Also, if a proof is available, one can check it by hand to make sure that the statement was inferred correctly (e.g., one may do this in order to prevent software errors which can be hidden in the automated prover and which can lead to incorrect results). Unlike most of the provers, our method gives for each proved statement its proof in a form of a sequence of formulas which can easily be understood by users.

The basic principle of our method can be described as follows. The *generator* which finds proofs and counterexamples consists of two coroutines: a *prover* which generates proofs and a *tester* which generates finite models as counterexamples. The generator accepts a set of hypotheses (interesting formulas) and a description of inference rules as its input. During the computation, the generator repeatedly switches between the two coroutines (prover and tester) and processes the set of hypotheses trying to classify each hypothesis into one of two sets: a *set of proved formulas* (if prover has found a proof of the formula) and a *set of disproved formulas* (if tester has found a counterexample against the formula). The generator

finishes either if each formula from the set of hypotheses has been classified in the previous sense or if it reaches a preset limit. Hence, a user supplies a set of hypotheses together with an inference system and the generator outputs a set of proved statements (along with their proofs), a set of disproved ones (along with the counterexamples), and a set of formulas which has not been classified. This is where out method differentiates from the other methods of automated deduction. The output of the generator can be further used. For instance, a user may inspect the formulas being proved/disproved/left unclassified, then modify the inference system (e.g. by adding new inference rules and axioms) and run the generator again to see how the sets of proved/disproved/unclassified formulas change.

The efficiency of our approach depends mainly on the method of generating of proofs. In this paper we present a preliminary method which is based on a *selective application of deduction rules* improved by a *reuse of previously proved formulas.* Our approach may resemble techniques of generating programs which are used in genetic programming [9]: an indispensable role is played by randomness supported by additional improvements. In this paper we focus on an improvement which is also inspired by human-like deduction, namely, we focus on a reuse of previously proved formulas. Needless to say, reuse of proved formulas is quite natural because if one tries to prove a statement, one usually builds a collection of (auxiliary) proved statements which are further used in the deduction to prove "more complex statements". We demonstrate by examples that the improvement of the basic method by a reuse of proved formulas has a considerable impact on the efficiency of the prover.

Our paper is organized as follows. In Section II we introduce the notions of provability and semantic entailment which will formalize notions we have used so far on an intuitive basis. Section III describes our basic technique of generating of proofs. In Section IV we present two case-studies: we discuss efficiency of the proposed method and its refinements. Section V concludes the paper and outlines a future research.

II. Provability and Semantic Entailment

In this section we present abstract notions of provability and semantic entailment which will be used in further sections. We set the notions rather general because we want all reasonable proof systems to fit in our framework. Our setting of basic notions is inspired by the abstract system for fuzzy logic which is due to Pavelka [13], see also [6]. Unlike Pavelka's system, the presented abstract system is more general and is not primarily aimed at fuzzy logic (although, proof systems of various fuzzy logics can be captured in our system).

We start by considering formulas. Throughout the paper, we denote by Fml a *set of all formulas in question.* Each element from the set Fml will be called a *formula.* Formulas will be denoted by $\varphi, \psi, \chi, \ldots$ At this point, no additional structure on Fml is assumed. That is, formulas are thought of as "abstract expressions" contained in Fml. Formulas are intended to formalize statements which we are concerned with. Of course, when considering particular proof systems, we first have to clarify Fml.

Next step is to formalize rules which allow us to infer (deduce) formulas from collections of other formulas. Given a set Fml of formulas, each partial map $R\colon Fml^n \to Fml$ will be called an *n-ary deduction rule* (*over Fml*). Deduction rules should be understood as follows. An n-ary deduction rule $R\colon Fml^n \to Fml$ either assigns to each n-tuple $\langle \varphi_1, \ldots, \varphi_n \rangle$ of formulas from Fml a formula $R(\varphi_1, \ldots, \varphi_n) \in Fml$ in which case we say that "R allows us to infer $R(\varphi_1, \ldots, \varphi_n)$ from $\varphi_1, \ldots, \varphi_n$", or R is not defined for $\langle \varphi_1, \ldots, \varphi_n \rangle$ (i.e., R cannot be used with formulas $\varphi_1, \ldots, \varphi_n$). In other words, by $R(\varphi_1, \ldots, \varphi_n) = \varphi$ we mean that from the input formulas $\varphi_1, \ldots, \varphi_n$ one gets the output formula φ using R. Deduction rules are often (informally) described using schemes such as

$$\frac{\Phi_1, \ldots, \Phi_n}{\Phi} \quad \text{or} \quad \text{"from } \Phi_1, \ldots, \Phi_n \text{ infer } \Phi\text{"},$$

saying, roughly speaking, if formulas $\varphi_1, \ldots, \varphi_n, \varphi$ match schemes $\Phi_1, \ldots, \Phi_n, \Phi$, then R is defined for $\varphi_1, \ldots, \varphi_n$, and $R(\varphi_1, \ldots, \varphi_n) = \varphi$. Let us stress again that such a description is informal (at least until we specify what "to match a scheme" means).

A nonempty set $\mathcal{R}$ of deduction rules over Fml will be called a *deductive system* (*over Fml*). Deductive systems are formalizations of collections of inference rules which are used to "infer new statements from other statements". In addition to the "inference rules" one usually considers some "axioms", i.e. statements (formulas) which are "taken for granted". Formally, there are two ways of looking at axioms. First, each axiom can be seen as a nullary deduction rule selecting from Fml a single formula $\varphi \in Fml$. Second, axioms are understood as designated formulas from Fml (which are assumed to be valid). Since the latter viewpoint is more convenient and more widely accepted, in addition to $\mathcal{R}$ we always consider a set $A \subseteq Fml$ which will be called a *set of axioms* (*over Fml*). In general, any subset $T \subseteq Fml$ will be called a *theory* (*over Fml*).

Now, assume we are given $\mathcal{R}$ (deductive system over Fml), and A (set of axioms over Fml). Let T be a theory over Fml. Each finite sequence $\delta_1, \ldots, \delta_k$ of formulas from Fml satisfying, for each $i = 1, \ldots, k$,

(i) $\delta_i \in A \cup T$, or

(ii) there are indices $i_1, \ldots, i_n < i$ such that for some n-ary deduction rule $R \in \mathcal{R}$: R is defined for $\langle \delta_{i_1}, \ldots, \delta_{i_n} \rangle$ and $R(\delta_{i_1}, \ldots, \delta_{i_n}) = \delta_i$,

is called a *proof* (*of δ_k*) *from T using $\mathcal{R}$ and A.* Formula $\varphi \in Fml$ is called *provable from T using $\mathcal{R}$ and A*, written $T \vdash^{\mathcal{R},A} \varphi$, if there exists a proof of φ from T using $\mathcal{R}$ and A. When $\mathcal{R}$ and A are clear from the context, we denote the fact $T \vdash^{\mathcal{R},A} \varphi$ by $T \vdash \varphi$ and say just "φ is provable from T". Our formalization of a proof is the usual one which is used in various logical calculi: a proof is a sequence of formulas each formula of which either is an axiom of belongs to the theory, or it is inferred by some deduction rule from previous formulas (i.e., from the formulas which are already proved). Hence, our notion of a proof corresponds well with the intuitive notion of a proof, where one "starts with axioms" and "draws conclusions" using inference rules. The following example shows particular deductive systems which are captured in our formal system.

Example 1: (a) The proof system of classical propositional logic [12] can be introduced in our formal setting as follows. First, we assume that we have a denumerable (countably infinite) set At of propositional symbols. The set Fml of all (propositional) formulas is defined as the least set satisfying (i) $At \subseteq Fml$, i.e. each propositional symbol is in Fml; and (ii) if $\varphi, \psi \in Fml$, then $\neg\varphi \in Fml$, and $(\varphi \to \psi) \in Fml$ (we assume that $\neg$ and $\to$ are the only symbols of logical connectives [12]). Furthermore, $\mathcal{R}$ contains a single binary rule R (*modus ponens*), which is defined for each $\varphi \in Fml$ and $(\varphi \to \psi) \in Fml$ so that $R(\varphi, (\varphi \to \psi)) = \psi$ (R is undefined otherwise). Set of axioms A is an infinite subset of Fml which is defined as follows:

$$\begin{aligned} A = &\{(\varphi \to (\psi \to \varphi)) \,|\, \varphi, \psi \in Fml\} \cup \\ &\{((\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))) \,| \\ &\qquad \varphi, \psi, \chi \in Fml\} \cup \\ &\{((\neg\varphi \to \neg\psi) \to (\psi \to \varphi)) \,|\, \varphi, \psi \in Fml\}. \end{aligned}$$

(b) As a second example, consider the equational logic [4], [14]. Denote by X a denumerable set of variables. Fix a collection F of function symbols (with the information about their arities). A set $T(X)$ of all terms is defined as follows: each $x \in X$ is a term; if $t_1, \ldots, t_n$ are terms and $f \in F$ is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term. An identity is any expression $t \approx t'$ where $t, t' \in T(X)$. The intended meaning of $t \approx t'$ is "t equals t'". Let Fml be the set of all identities. Note that Fml is determined by the chosen collection of function symbols F. For instance, if F consists of a binary symbol $\circ$, a unary symbol $^{-1}$, and a nullary symbol e, then formulas in Fml ($x \circ e \approx x$, $x^{-1} \approx (e \circ y)^{-1} \circ e, \ldots$) are exactly the identities in language of group theory (caution: the fact that $t \approx t'$ is an identity written in language of groups does not necessarily mean that $t \approx t'$ is true in all groups). For $s, t \in T(X)$ and $x \in X$, let $t(x/s)$ denote term resulting from t by replacing all occurrences of the variable x in t by s. Deduction rules of equational logic (informally written) are the following [4]:

(Sym) from $t \approx t'$ infer $t' \approx t$;
(Tra) from $t \approx t'$ and $t' \approx t''$ infer $t \approx t''$;
(Rep) from $t \approx t'$ infer $r \approx r'$, where r contains t as a subterm and r' results from r by replacing the subterm t by t';
(Sub) for each $x \in X$ and $s \in T(X)$, from $t \approx t'$ infer $t(x/s) \approx t'(x/s)$.

(Sym)–(Sub) can be formalized by deduction rules of the from $R\colon Fml^n \to Fml$. For instance, (Sym) is a rule $R\colon Fml \to Fml$ such that, for each $t \approx t'$, $R(t \approx t') = t' \approx t$. Note also that schemes (Rep) and (Sub) are formalized by infinitely many rules. For example, scheme (Sub) can be expressed by a set $\{R_{x,s} \,|\, x \in X, s \in T(X)\}$ of rules with each $R_{x,s}$ defined by $R_{x,s}(t \approx t') = t(x/s) \approx t'(x/s)$. Take any variable $x \in X$. Define $A = \{x \approx x\}$. That is, the set of axioms of equational logic consists of a single formula, saying "each element is equal to itself".

We now turn our attention to the semantic entailment, i.e. to the entailment which is based on the notion of truth. In order to consider "truth of formulas" (i.e., their semantic validity), we need to introduce a semantic component in which we evaluate formulas and describe the evaluation. Because of the generality of our approach, we introduce semantic structures as subsets of Fml, meaning that a particular semantic structure determines (somehow) which formulas from Fml are true (in that structure). A precise definition follows.

Each system $\mathcal{S} \subseteq 2^{Fml}$ will be called a *semantics for* Fml. That is, a semantics for Fml is a subset of the power set of Fml. Each $M \in \mathcal{S}$ is thus a set of formulas from Fml. We call each $M \in \mathcal{S}$ a (*semantic*) *$\mathcal{S}$-structure* (shortly, a *structure*). The intended meaning of $\varphi \in M$ is "φ is true in M"; $\varphi \notin M$ means "φ is not true (is false) in M". An $\mathcal{S}$-structure M is called an *$\mathcal{S}$-model* (shortly, a *model*) *of a theory* $T \subseteq Fml$ if $T \subseteq M$. Taking into account our interpretation of $\varphi \in M$, the fact that M is a model of T means that each formula from T is true in M, which is the usual way of defining the notion of a model, see e.g. [12].

Using $\mathcal{S}$-models of T, we can define semantic entailment. Assume we are given a semantics $\mathcal{S} \subseteq 2^{Fml}$. Formula $\varphi \in Fml$ *$\mathcal{S}$-semantically follows from a theory* $T \subseteq Fml$, written $T \models^{\mathcal{S}} \varphi$, if φ is true in each $\mathcal{S}$-model of T. Thus, $T \models^{\mathcal{S}} \varphi$ iff, for each $M \in \mathcal{S}$ such that $T \subseteq M$, we have $\varphi \in M$. If $\mathcal{S}$ is clear from context, we write $T \models \varphi$ (and say φ semantically follows from T) instead of writing $T \models^{\mathcal{S}} \varphi$ (and saying φ $\mathcal{S}$-semantically follows from T).

Now, we have specified two types of entailment: a *syntactic one* (called provability and denoted $\vdash$) and a *semantic one* (denoted $\models$). Syntactic entailment is based on a particular manipulation with formulas (constructing of proofs) disregarding any interpretation of the formulas whereas semantic entailment is based on truth in all models. Any useful logical calculus should have a reasonable connection between these two types of entailment. A logical calculus should at least be sound: each formula provable from a theory T should be true in all models of T. Soundness ensures that one cannot infer more than it is "actually true". A stronger property, completeness (of a logical calculus), says that semantic consequences of a theory are exactly the formulas which are provable from that theory. Soundness and completeness can be formulated in our abstract setting as follows.

Let $\mathcal{R}$ be a deductive system over Fml, $A \subseteq Fml$ be a set of axioms, and $\mathcal{S} \subseteq 2^{Fml}$ be a semantics. Then, *$\mathcal{R}$ with A are sound over $\mathcal{S}$* if, for each theory $T \subseteq Fml$ and each formula $\varphi \in Fml$, we have

$$\text{if } T \vdash^{\mathcal{R},A} \varphi, \text{ then } T \models^{\mathcal{S}} \varphi. \tag{1}$$

Observe that (1) is equivalent to

$$\text{if } T \not\models^{\mathcal{S}} \varphi, \text{ then } T \not\vdash^{\mathcal{R},A} \varphi, \tag{2}$$

saying: "if φ is false in some $\mathcal{S}$-model of T, then φ is not provable from T using $\mathcal{R}$ and A". Therefore, if we assume $\mathcal{R}$ and A being sound for $\mathcal{S}$, in order to show that there is no proof of φ from T, it suffices to find one $\mathcal{S}$-model M of T in which φ is false (i.e., $\varphi \notin M$). Hence, due to soundness, we are able to "disprove formulas" by finding "counterexamples", this technique is also widely used on the intuitive level of deduction.

Completeness is formulated analogously: $\mathcal{R}$ *with* A *are complete over* $\mathcal{S}$ if, for each $T \subseteq Fml$ and $\varphi \in Fml$,

$$T \vdash^{\mathcal{R},A} \varphi \quad \text{iff} \quad T \models^{\mathcal{S}} \varphi. \tag{3}$$

The following example contains semantics for proof systems described in Example 1.

Example 2: (a) Consider set Fml of formulas, deduction rules $\mathcal{R}$, and set of axioms A from Example 1 (a). For each $M \subseteq At$ (recall that At is the set of all propositional symbols), denote by $\overline{M}$ the subset of Fml which is uniquely given by the following conditions: (i) $M \subseteq \overline{M}$; (ii) $\neg\varphi \in \overline{M}$ iff $\varphi \notin \overline{M}$; and (iii) $(\varphi \to \psi) \in \overline{M}$ iff $\varphi \notin \overline{M}$ or $\psi \in \overline{M}$. Put $\mathcal{S} = \{\overline{M} \,|\, M \subseteq At\}$. From the well-known results on completeness of propositional logic [12] we get that $\mathcal{R}$ with A are complete over $\mathcal{S}$-semantics.

(b) Take Fml, $\mathcal{R}$, and A from Example 1 (b). Recall that Fml is given by a collection F of functions symbols with their arities. An algebra $\mathbf{M} = \langle M, F^{\mathbf{M}}\rangle$ of type F consists of a nonempty universe set M, and a collection $F^{\mathbf{M}}$ of n-ary functions such that for each n-ary symbol $f \in F$, $F^{\mathbf{M}}$ contains a function $f^{\mathbf{M}} \colon M^n \to M$. By an $\mathbf{M}$-valuation is meant any map $v \colon X \to M$ assigning elements of M to variables from X. For each term t, we define the value $||t||_{\mathbf{M},v}$ of t in $\mathbf{M}$ under v as follows: (i) if $t \in X$, then $||t||_{\mathbf{M},v} = v(t)$; (ii) if t is $f(t_1, \ldots, t_n)$, then $||t||_{\mathbf{M},v} = f^{\mathbf{M}}(||t_1||_{\mathbf{M},v}, \ldots, ||t_n||_{\mathbf{M},v})$. Moreover, for $\mathbf{M}$ and v put $S_{\mathbf{M},v} = \{t \approx t' \in Fml \,|\, ||t||_{\mathbf{M},v} = ||t'||_{\mathbf{M},v}\}$. Now, define semantics $\mathcal{S}$ for Fml by

$$\mathcal{S} = \{S_{\mathbf{M},v} \,|\, \mathbf{M} \text{ is algebra of type } F, \; v \colon X \to M \text{ is } \mathbf{M}\text{-valuation}\}. \tag{4}$$

Due to the results on completeness of equational logic which are due to Birkhoff [4], [14], we have that $\mathcal{R}$ with A are complete over the $\mathcal{S}$-semantics defined by (4).

III. Basic Technique of Generating of Proofs

The basic idea of the generator has been briefly described in Section I. Let us recall that the generator consists of two (independent) coroutines: a *prover* and a *tester.* From the logical point of view, the aim of the prover is to generate proofs given a deductive system $\mathcal{R}$, a set of axioms A, and a theory T as its input. The tester generates finite structures from a given semantics $\mathcal{S}$ which are $\mathcal{S}$-models of T (we will not describe this procedure in detail here).

The generator accepts as its input a deductive system $\mathcal{R}$ (representing the rules of inference), a set of axioms A, a user supplied theory T, a semantics $\mathcal{S}$, and a set H of *hypotheses.* We always assume that $\mathcal{R}$ with A are sound over $\mathcal{S}$ in order to be able to "disprove formulas" by "counterexamples". The set H represents a collection of statements the user is interested in. The role of generator is to coordinate prover and tester to split the input set H of hypotheses into three sets:

1) a *set of proved formulas*, denoted by P,
2) a *set of disproved formulas*, denoted by D,
3) a set of formulas from H for which the generator cannot decide whether they belong to P or D. We call this set a set of *unclassified formulas* and denote it by H.

At the beginning of a computation, the sets P and D are empty, and the set of hypotheses H coincides with the set of unclassified formulas. After its invocation, the generator tries to classify each hypothesis from H either as a provable formula (set P) or as a formula which can be disproved by a counterexample (set D). In detail, for each $\varphi \in H$, the generator intends to show $T \vdash^{\mathcal{R},A} \varphi$ or $T \not\models^{\mathcal{S}} \varphi$ (and thus $T \not\vdash^{\mathcal{R},A} \varphi$ due to soundness). The fact $T \vdash^{\mathcal{R},A} \varphi$ follows immediately if the prover finds any proof $\delta_1, \ldots, \delta_k$ from T using $\mathcal{R}$ and A such that $\varphi \in \{\delta_1, \ldots, \delta_k\}$. On the other hand, if tester finds an $\mathcal{S}$-model M of T such that $\varphi \notin M$, we get that $T \not\models^{\mathcal{S}} \varphi$ and, consequently, φ can be categorized as "not provable".

The generator alternate between two basic phases of computation. First, a phase in which it uses prover to generate proofs and shows that formulas from H are provable. Second, a phase in which the generator uses tester to generate models and shows that formulas from H are not provable. The process halts if H (the set of unclassified formulas, i.e. the set of "remaining" hypotheses) is empty (each formula has been classified) or if the generator reaches a given limit (e.g., a maximal number of iterations). Thus, the generator in its basic setting can be summarized as the following procedure:

0) Set P and D to $\emptyset$. Initialize variables m (number of prover loops in each iteration), n (number of tester loops in each iteration), and k (initial length of proofs). Go to step 1.
1) If $H = \emptyset$ (no hypothesis left) or if a limit condition is reached, then halt and output P and D; else go to step 2.
2) Repeat m times:
   a) Use prover to generate a proof $\varphi_1, \ldots, \varphi_k$ of length k.
   b) For each formula φ_i $(i = 1, \ldots, k)$ do:
      if $\varphi_i \in H$, then remove φ_i from H and add φ_i to P;
      if $H = \emptyset$, go to step 1.
3) Repeat n times:
   a) Use tester to generate an $\mathcal{S}$-model M of T.
   b) For each formula $\varphi \in H$ do:
      if $\varphi \notin M$, then remove φ from H and add φ to D;
      if $H = \emptyset$, go to step 1.
4) Modify values of m, n, and k and go to step 1.

In the previous algorithm, we have tacitly used variables m, n, and k whose role is to parameterize the computation process. The generator can be run with low values of k and with each iteration the value of k can be increased. Hence, the prover produces short proofs first and then it tries to produce longer proofs. Of course, there may be several reasonable ways to set and modify m, n, and k. The basic schema of the the generator as described above tries to follow typical phases of a human-like inference, see the discussion in Section I. Note also that in step 2b, with each φ_i being added to P we can store its proof $\varphi_1, \ldots, \varphi_i$.

We now briefly describe the prover which is a kernel of the automated deduction. For a given k (length of a proof), the prover generates a proof $\varphi_1, \ldots, \varphi_k$ of length k as follows:

0) Set i to 1. Go to step 1.
1) If $i > k$, halt and return $\varphi_1, \ldots, \varphi_k$; else go to step 2.
2) If $i = 1$, then go to step 2a,
   otherwise choose randomly between the following steps:
   a) randomly pick $\varphi_i \in A \cup T$ and go to step 3.

b) select an n-ary rule $R \in \mathcal{R}$ and formulas $\varphi_{i_1}, \ldots, \varphi_{i_n}$ where $i_1, \ldots, i_n < i$. If R is defined for $\varphi_{i_1}, \ldots, \varphi_{i_n}$ then put $\varphi_i = R(\varphi_{i_1}, \ldots, \varphi_{i_n})$ and goto step 3. If R is not defined for $\varphi_{i_1}, \ldots, \varphi_{i_n}$ then go to the beginning of step 2.

3) Increase i and go to step 1.

The above-described procedure can be further improved. For instance, we should skip formulas which are already present in the proof. That is, if φ_i is already present between $\{\varphi_1, \ldots, \varphi_{i-1}\}$ we may go to the beginning of step 2 instead of the prolongation of the proof by φ_i (multiple occurrences of formulas in proofs do not yield anything useful). Let us note that for a particular deductive system $\mathcal{R}$ with axioms A a user should define how the prover selects deduction rules and axioms (for simplicity, we assume that the selection is random according to priorities of rules and axioms).

Another improvement of the method whose efficiency will be discussed in further sections is a reuse of previously proved formulas (see Section I). Notice that in step 2 of the prover we may add an alternative 2c saying "select $\varphi_i \in P$ and go to step 3", where P is the set of proved formulas maintained by the generator. This represents a reuse of formulas in P. Notice that adding formulas from P to a proof may lead to a sequence of formulas which is no longer a proof. Nevertheless, for such sequences we can always construct a proof by replacing each formula $\vartheta \in P$ occurring in the sequence by the proof of ϑ (we omit details). From the viewpoint of formulas from P we have three basic options:

1) Do not reuse previously proved formulas.
2) Reuse previously proved formulas and make an equal choice between them (i.e., all proved formulas are assigned the same priority for their future use).
3) Reuse previously proved formulas according to their "interestingness" (interesting formulas, e.g. formulas which were hard to prove, are used more frequently).

In Section IV we demonstrate that reuse methods 2) and 3) increase performance of the generator compared to 1). On the other hand, for various criterions of interestingness, a series of tests does not show any significant difference between 2) and 3), see also Section IV.

IV. Two Case Studies

In this section we present two basic examples of how the general procedure can be used to find proofs and counterexamples. In each subsection we first specify the proof system, i.e. deductive system and set of axioms, and the semantics. Then we comment on the efficiency of the method and its refinements.

A. Logic of Galois Connections with Hedges

Galois connections with hedges naturally appear in several areas of data analysis [1], [2], [3]. Namely, in [1], [2] we introduced Galois connections with hedges as pairs of mappings induced by object-attribute data tables with fuzzy attributes. Fixed points of such connections can be interpreted as clusters found in the data. Paper [2] shows further applications of Galois connections with hedges in generating of IF-THEN rules from data. When exploring properties of Galois connections with hedges, we often constructed proofs in which we used expressions of the form $B^{\cdots} \subseteq B^{\cdots}$, where "..." stand for sequences of symbols $'$ (symbol denoting so-called "prime operator") and * (symbol denoting so-called "star operator"). Each expression $B^{\cdots} \subseteq B^{\cdots}$ is an assertion on its own and should be proved (or disproved). Since the proofs of expressions $B^{\cdots} \subseteq B^{\cdots}$ turned our to be routine and purely combinatorial, it became in handy to automate the deduction by a computer. This was, in fact, the original motivation for the generator described in this paper. In this section we introduce the deductive system and comment on the efficiency of the methods proposed in Section III.

We now specify formulas, deductive system, axioms and semantics for Galois connection with hedges. Let Σ be an alphabet (a finite set of symbols) defined by $\Sigma = \{*, \prime\}$. Strings over Σ (i.e., finite sequences of symbols from Σ) will be denoted by $\alpha, \beta, \ldots$, length of a string α will be denoted by $|\alpha|$. Let Σ_E (Σ_O) be sets of all strings over Σ which have even (odd) number of occurrences of the prime. We fix symbol B the intuitive interpretation of which is to "denote collections of attributes" (the role of B will become apparent later). The set Fml of formulas is defined by $Fml = \{B^\alpha \subseteq B^\beta \mid \alpha, \beta \in \Sigma_E \text{ or } \alpha, \beta \in \Sigma_O\}$. Caution: at this point, "$\subseteq$" which occurs in each formula is just a symbol (it has nothing to do with the subsethood relation). For each formula $B^\alpha \subseteq B^\beta$ we can consider its *complexity* which is a non-negative integer defined by $\max(|\alpha|, |\beta|)$, i.e. the complexity of $B^\alpha \subseteq B^\beta$ is the maximum of lengths of strings α and β. For instance, complexity of $B^{*\prime\prime *\prime} \subseteq B^{*\prime}$ is 5, complexity of $B \subseteq B$ is 0, etc.

Our deductive system $\mathcal{R}$ consists of five unary deduction rules $R_1, \ldots, R_5$ which are all defined for any $B^\alpha \subseteq B^\beta$ (i.e., each of $R_1, \ldots, R_5$ is a total map sending Fml to Fml). The rules are defined as follows:

$$R_1(B^\alpha \subseteq B^\beta) = B^{\alpha *} \subseteq B^\beta, \tag{5}$$

$$R_2(B^\alpha \subseteq B^\beta) = B^{\alpha *} \subseteq B^{\beta *}, \tag{6}$$

$$R_3(B^\alpha \subseteq B^\beta) = B^{\beta \prime} \subseteq B^{\alpha \prime}, \tag{7}$$

$$R_4(B^\alpha \subseteq B^\beta) = B^\alpha \subseteq B^{\beta \prime\prime}, \tag{8}$$

$$R_5(B^\alpha \subseteq B^\beta) = B^\alpha \subseteq B^{\beta \prime * \prime}. \tag{9}$$

Put $A = \{B \subseteq B\}$, i.e. the set of axioms contains a single formula $B \subseteq B$. We now have defined all the syntactic components which enable us to consider the corresponding provability relation $\vdash^{\mathcal{R},A}$.

The semantics of Galois connections with hedges is connected with data tables with fuzzy attributes. We present only a brief introduction of the semantics because a detailed description is outside the scope of this paper. Details can be found in [1], [2]. Let $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \Rightarrow, v, 0, 1 \rangle$ be a complete residuated lattice with hedge v [6], [7]. Note the $\otimes$ and $\Rightarrow$ are truth functions that interpret fuzzy logical connectives "conjunction" and "implication"; v can be seen as an interpretation of connective "very true" [7]. Denote by $\leq$ the lattice order induced by $\mathbf{L}$. A fuzzy set (or $\mathbf{L}$-set) C in universe U is any map $C \colon U \to L$, $C(u)$ being interpreted as "the degree to which u belongs to C". A binary fuzzy relation ($\mathbf{L}$-relation) R between U and V is any map $R \colon U \times V \to L$,

$R(u, v)$ being interpreted as "the degree to which u and v are R-related". A fuzzy set $C : U \to L$ is a subset of a fuzzy set $D : U \to L$, written $C \subseteq D$, if $C(u) \leq D(u)$ is true for each $u \in U$. Let X and Y be sets of objects and attributes, respectively, I be a binary fuzzy relation between X and Y. The triplet $\langle X, Y, I\rangle$ is called a *data table with fuzzy attributes*. $\langle X, Y, I\rangle$ represents a table which assigns to each $x \in X$ and each $y \in Y$ a truth degree $I(x, y) \in L$ to which object x has attribute y. For fuzzy sets $C : X \to L$, $D : Y \to L$ (i.e. C is a fuzzy set of objects, D is a fuzzy set of attributes), we define fuzzy sets $C' : Y \to L$, $C^* : X \to L$, $D' : X \to L$, and $D^* : Y \to L$ by

$$\begin{aligned} C'(y) &= \textstyle\bigwedge_{x\in X}\big(C(x) \Rightarrow I(x, y)\big),\\ (C^*)(x) &= v(C(x)),\\ D'(x) &= \textstyle\bigwedge_{y\in Y}\big(D(y) \Rightarrow I(x, y)\big),\\ (D^*)(x) &= v(D(x)). \end{aligned}$$

Described verbally, C' is the collection of all attributes shared by all objects from C, D' is the collection of all objects sharing all attributes from D, C^* and D^* can be seen as collections of objects and attributes which are very contained in C and D, respectively. Observe that for any $D \in \mathbf{L}^Y$ and any sequence $\alpha = a_1 a_2 \cdots a_m$ over Σ, we can consider a fuzzy set D^α (of objects or attributes) defined as $(\cdots((D^{a_1})^{a_2})\cdots)^{a_m}$. For instance, $D'^{*\prime} = ((D')^*)'$ is a fuzzy set of attributes, $D'^{*\prime**\prime*}$ is a fuzzy set of objects, etc.

Given $\mathbf{L}$, $\langle X, Y, I\rangle$ with $I : X \times Y \to L$, and $D \in \mathbf{L}^Y$ we define $S_{\mathbf{L},I,D} \subseteq Fml$ by

$$S_{\mathbf{L},I,D} = \{B^\alpha \subseteq B^\beta \in Fml \,|\, D^\alpha \subseteq D^\beta\}.$$

i.e. a formula $B^\alpha \subseteq B^\beta$ belongs to $S_{\mathbf{L},I,D}$ iff the fuzzy set D^α is a subset of the fuzzy set D^β. Observe that both D^α and D^β either are fuzzy sets of objects or fuzzy sets of attributes (this is ensured by the odd/even number of primes in the strings α and β). The semantics $\mathcal{S}$ is defined as a set of all $S_{\mathbf{L},I,D}$ ($\mathbf{L}$, $\langle X, Y, I\rangle$, and D are arbitrary). Using properties of $\mathbf{L}$, one can check that $\mathcal{R}$ and A are sound over $\mathcal{S}$. Thus, we can apply the inference mechanism described in Section III.

Tests and Results The algorithm for generating of proofs was tested on several interesting sets of formulas from Fml according to complexity of formulas. For each integer $1 \leq n \leq 9$, we picked from Fml formulas whose complexity was at most n. In order to eliminate formulas which are not "interesting", we skipped formulas of the form $B^\alpha \subseteq B^\alpha$ which are evidently provable by the repeated use of (6) and (7); formulas which contain two or more stars in a row (e.g., $B'^{**} \subseteq B'$) because of the idempotency of v, see [1], [2]; and formulas which contain three or more primes in a row (e.g., $B^{*\prime\prime\prime} \subseteq B^{*\prime}$) because $'''$ can be equivalently replaced by $'$, see [1], [2].

For each set of interesting formulas of a given complexity, we repeatedly ran the generator in order to get estimations of the average time usage. Note that the "prover" coroutine of the generator was set to use exactly the deductive system $\mathcal{R}$ and the set of axioms A as introduced above. On the other hand, the "tester" coroutine was set to use only a subset $\mathcal{S}'$ of the above-defined $\mathcal{S}$. Namely, $\mathcal{S}' = \{S_{\mathbf{L},I,D} \,|\, \mathbf{L}, I, \text{ and } D \text{ are finite}\}$.

1. $B \subseteq B$	axiom	1. $B \subseteq B$	axiom
2. $B^* \subseteq B^*$	(6) on 1.	2. $B^* \subseteq B^*$	(6) on 1.
3. $B^{*\prime} \subseteq B^{*\prime}$	(7) on 2.	3. $B^* \subseteq B^{*\prime\prime}$	(8) on 2.
4. $B^{*\prime} \subseteq B^{*\prime\prime*\prime}$	(9) on 3.	4. $B^* \subseteq B^{*\prime\prime*}$	(6) on 3.
		5. $B^{*\prime\prime*\prime} \subseteq B^{*\prime}$	(7) on 4.

Fig. 1. Generated proofs of $B^{*\prime} \subseteq B^{*\prime\prime*\prime}$ (left) and $B^{*\prime\prime*\prime} \subseteq B^{*\prime}$ (right).

This restriction is, of course, due to the impossibility to represent general infinite structures by a computer. Nevertheless, $\mathcal{S}'$ turns out to be sufficient for all the interesting formulas we considered: each formula either was proved using $\mathcal{R}$ and A or, a counterexample was found in $\mathcal{S}'$.

For instance, for $n = 3$, there are 43 interesting formulas fulfilling our constraints, 17 of which are provable (from the empty theory), and the rest (26 formulas) are not provable (for each of the 26 formulas a counterexample was found). For $n = 3$, the provable formulas are the following:

$$\begin{array}{llll} B \subseteq B'', & B^* \subseteq B^{*\prime\prime}, & B'^* \subseteq B', & B^{*\prime*} \subseteq B^{*\prime},\\ B \subseteq B'^{*\prime}, & B^* \subseteq B'^{*\prime}, & B'^* \subseteq B^{*\prime}, & B^{*\prime\prime} \subseteq B'',\\ B^* \subseteq B, & B^* \subseteq B''^*, & B'^* \subseteq B^{*\prime*}, & B^{*\prime\prime} \subseteq B'^{*\prime},\\ B^* \subseteq B'', & B' \subseteq B^{*\prime}, & B'' \subseteq B'^{*\prime}, & B''^* \subseteq B'',\\ B''^* \subseteq B'^{*\prime}. & & & \end{array}$$

In this case ($n = 3$), the running time of generator was approximately 0.08 seconds[1]. During that time each input formula was classified either as to be provable, or not. Also note that for $n = 3$, there was almost no difference in efficiency between particular *methods of reuse* (of previously proved formulas), see Section III. For each of the methods, the average time consumed by the generator was practically the same. For illustration, Fig. 1 contains proofs of formulas $B^{*\prime} \subseteq B^{*\prime\prime*\prime}$ and $B^{*\prime\prime*\prime} \subseteq B^{*\prime}$ as they were generated by the machine. These two assertions together yield an important property of Galois connections with hedges which has been used, e.g., in [3].

As the complexity of formulas grows, the performance of the generator becomes to diverge according to the method of reuse applied by the "prover" coroutine. For example, in case of $n = 9$, there are 3122 interesting formulas, 1255 of which are provable, and the rest (1867 formulas) are not. If the prover does not use previously proved formulas at all, the average running time of the generator is 1020 seconds. On the other hand, if the prover constructs proofs also using the previously proved formulas, the running time is cut down to approximately 24 seconds. Thus, the reuse of formulas has a considerable impact on the efficiency of prover. On the other hand, the tests have shown that methods for the selection of formulas to be used in a proof produce almost the same results. For this particular example, the method which prefers formulas with longer proofs was slightly more efficient (23 seconds for $n = 9$) than the random selection (25 seconds for $n = 9$).

Table I summarizes average running time of the generator according to the complexity of formulas (row labeled "hypotheses" contains number of generated input formulas; row

[1]All tests have been done on a standard PC (3 GHz CPU, 2 GB RAM), the algorithms were implemented using the Bigloo Scheme compiler.

TABLE I

PROOFS IN LOGIC OF GALOIS CONNECTIONS WITH HEDGES

	complexity of formulas								
	1	2	3	4	5	6	7	8	9
hypotheses	2	12	43	102	220	450	894	1682	3122
proved	1	6	17	37	82	175	335	642	1255
disproved	1	6	26	65	138	275	559	1040	1867
avg. time 1	0.01	0.02	0.08	0.2	0.3	1.5	16	109	1020
avg. time 2	0.01	0.02	0.08	0.2	0.3	0.8	3	8	24

labeled "proved" contains number of proved formulas; row labeled "disproved" contains number of formulas which were disproved by counterexamples; rows labeled "avg. time 1/2" contain average running times of the generator without/with the reuse of proved formulas). A database of proved formulas (with complexity up to 9) and their proofs (written as in Fig. 1) is freely available, see Section V.

B. Automated Deduction in Equational Logic

In this section, we are going to use the generator to prove identities describing properties of groups. We choose this example because (i) groups are well-known algebraic structures and (ii) groups are can be determined by sets of identities [14], i.e. we can use deductive system $\mathcal{R}$, sets of axioms A, and semantics $\mathcal{S}$ from Example 1 (b) and Example 2 (b). In particular, F (collection of function symbols) contains a binary function symbol $\circ$ (denoting group operation), a unary function symbol $^{-1}$ (denoting inversion), and a nullary function symbol (constant) e which is a symbol denoting the neutral element of a group. The set of formulas (identities) is then uniquely given. Since we are interested in proofs from group theory, we fix a theory $T \subseteq Fml$ as follows:

$$\begin{aligned} T = \{&x \circ (y \circ z) \approx (x \circ y) \circ z, \\ &x \circ e \approx x, e \circ x \approx x, \\ &x \circ x^{-1} \approx e, x^{-1} \circ x \approx e\}, \end{aligned} \tag{10}$$

i.e. T consists of the basic "group identities" [5] describing properties of $\circ$, $^{-1}$, and e. Let $\mathcal{R}$, A, and $\mathcal{S}$ be then defined as in Example 1 (b) and Example 2 (b). As a particular application of the Birkhoff's theorem [4], [14] we get that $t \approx t'$ is provable from T defined by (10) using $\mathcal{R}$ and A iff $t \approx t'$ is true in each group, which holds true iff $t \approx t'$ is true in each $\mathcal{S}$-model of T, see Example 2 (b). Thus, we have $\mathcal{R}$ and A which are complete (and thus sound) for $\mathcal{S}$, i.e., we may run the generator to infer formulas from T. Before that, we adjust the deductive system a little to make it more convenient for the automated deduction.

First, we represent each identity $t \approx t'$ by a set $\{t, t'\}$ of terms. Obviously, $|\{t, t'\}| = 1$ iff $t = t'$, i.e. iff $t \approx t'$ is a trivial identity. The new representation of identities will allow us to simplify deduction rules. Some of the new deduction rules will be formulated using general substitutions and unifiers. Since these notions are well known from logic programming [11], we omit their description (see [11] for details). We drop rule (Sym) which is no longer needed and we also drop the axiom $x \approx x$. The other rules will be modified as follows:

(Tra) from $\{t, t'\}$ and $\{s, s'\}$ infer $\{t'\theta, s'\theta\}$ provided that θ is the most general unifier of t and s;

(Con) from $\{s_1, s'_1\}, \ldots, \{s_k, s'_k\}$ infer $\{t, t'\}$ such that $t = f(t_1, \ldots, t_n)$, $t' = f(t'_1, \ldots, t'_n)$, f is an n-ary function symbol and, for each index $i = 1, \ldots, n$, either $|\{t_i, t'_i\}| = 1$ or there is index $j \in \{1, \ldots, k\}$ such that $\{t_i, t'_i\} = \{s_j, s'_j\}$;

(Sub) from $\{t, t'\}$ infer $\{t\theta, t'\theta\}$ for any substitution θ.

Finally, we put $A = \emptyset$ (no axioms). Note that the new rules still can be represented by maps of the from $R\colon Fml^n \to Fml$ (we omit details due to lack of space). Observe that the original transitivity rule (Tra), see Example 1 (b), allowed us to infer $t \approx t''$ from $t \approx t'$ and $t' \approx t''$, i.e., from two formulas with a common term t'. From the point of view of automated deduction, we want each rule to be defined for as much input formulas as possible because we want be able to use at each step of the inference as much rules as possible. The old (Tra) is too restrictive, because it can only be used for two identities sharing the same term. On the other hand, the modified (Tra) allows us to infer a new identity from other identities provided that the identities contain terms which are unifiable [11]—this is much weaker a condition.

The following assertion shows that our modified deductive system is equivalent (up to the different formalization of identities) to the former one.

Theorem 1: *Let T be a set of identities. Furthermore, put $\lceil T \rceil = \{\{s, s'\} \,|\, s \approx s' \in T\}$. Then, an identity $t \approx t'$ is provable from T using the original Birkhoff system iff either $t \approx t'$ is trivial* (i.e., $|\{t, t'\}| = 1$) or $\lceil T \rceil \vdash^{\mathcal{R},A} \{t, t'\}$.

Proof: Due to the limited scope of this paper, we postpone the proof to the full version of the paper. ■

Tests and Results As in subsection IV-A, we first generated a set of interesting formulas. This time, we picked 4000 identities up to a given complexity (complexity of identities is defined in much the same way as in case of formulas used in subsection IV-A, we omit details). Then we ran the generator and compared its efficiency according to various settings. The "prover" coroutine was set to use the modified deductive system described above, while the "tester" coroutine was set to use a subset of the semantics $\mathcal{S}$ (we picked only those $S_{\mathbf{M},v}$'s where $\mathbf{M}$ is finite and took the advantage of the fact that each group is isomorphic to a subgroup of some permutation group). From the 4000 identities being generated, only 181 were proved by the generator, and the rest (3819 identities) were disproved by counterexamples (found in finite groups).

The average running time of the generator varied significantly depending on particular settings (i.e., settings of random selection of deduction rules, particular method of reuse of previously proved formulas, etc.). However, during all the tests, a small set of identities turned out to be critical: all formulas in that set were provable from our initial theory (10), but the generator needed a large amount on time compared to the time spent on generating proofs of the other formulas. The critical set contained the following formulas:

1. $\{e \circ x, x\}$ axiom
2. $\{x \circ e, x\}$ axiom
3. $\{x \circ x^{-1}, e\}$ axiom
4. $\{y \circ e, y \circ (x \circ x^{-1})\}$ (Con) on 3.
5. $\{x \circ (y \circ z), (x \circ y) \circ z\}$ axiom
6. $\{e \circ x^{-1^{-1}}, (x \circ x^{-1}) \circ x^{-1^{-1}}\}$ (Con) on 3.
7. $\{e \circ x^{-1^{-1}}, x \circ (x^{-1} \circ x^{-1^{-1}})\}$ (Tra) on $6., 5.$
8. $\{e \circ z^{-1^{-1}}, z \circ (z^{-1} \circ z^{-1^{-1}})\}$ (Sub) on 7.
9. $\{z \circ (z^{-1} \circ z^{-1^{-1}}), z^{-1^{-1}}\}$ (Tra) on $8., 1.$
10. $\{y^{-1^{-1}}, y \circ e\}$ (Tra) on $9., 4.$
11. $\{x^{-1^{-1}}, x\}$ (Tra) on $10., 2.$

Fig. 2. Generated proof of $x \approx x^{-1^{-1}}$

$$x \approx x^{-1^{-1}}, \quad x \circ e^{-1} \approx x^{-1^{-1}}, \quad (e \circ x) \circ e \approx x^{-1^{-1}},$$
$$x \circ e \approx x^{-1^{-1}}, \quad e \circ (x \circ e) \approx x^{-1^{-1}}, \quad (e \circ e) \circ x \approx x^{-1^{-1}},$$
$$e \circ x \approx x^{-1^{-1}}, \quad e \circ (e \circ x) \approx x^{-1^{-1}}, \quad e^{-1} \circ x \approx x^{-1^{-1}},$$
$$x \circ (e \circ e) \approx x^{-1^{-1}}, \quad (x \circ e) \circ e \approx \approx x^{-1^{-1}}.$$

If the "prover" was set to reuse previously proved formulas, it took only 8 seconds to prove/disprove all formulas but the critical ones. Then, it took approximately 20 hours (the average time varies depending on further settings) to prove at least one of the critical formulas. After any of the critical formulas was proved, the rest (of the critical ones) was proved in mere seconds. An example of a generated proof of one of the critical formulas is in Fig. 2. Hence, during the processing of the input formulas, there is a long "gap" after the non-critical formulas are proved/disproved and before the generator finds a proof of a critical one. This is, in fact, interesting because it tells us that the law of double inversion $x \approx x^{-1^{-1}}$ which is essential to prove each of the critical formulas (and does not appear in the non-critical ones) is much harder to prove than other properties described by the generated formulas. Knowing that $x \approx x^{-1^{-1}}$ is critical and that it is provable from (10), we can add $x \approx x^{-1^{-1}}$ to (10) in order to speed up the generator and without changing the set of syntactic consequents of (10). After the addition of $x \approx x^{-1^{-1}}$, the generator processes all the 4000 formulas with the average running time of 8 seconds. A database of all the proofs is available, see Section V.

In this subsection we have used the group theory mainly for demonstration purposes. One may argue that in case of group theory, there are efficient ways to decide whether a given identity follows from a set of identities, making thus the prover superfluous. For instance, one can use the Knuth-Bendix procedure [10] to transform T defined by (10) into a terminating and confluent term-rewriting system [8], [14] which can be further used to decide whether an equality is provable from T. Note, however, that for most equational theories one cannot proceed this way in which case our generator may be helpful.

V. Conclusions and Future Research

We showed a general method of generating of proofs and discussed efficiency of a reuse of proved formulas. The main benefit of the generator is that it can help experts explore consequences from given sets of formulas. This can be useful from both the theoretical and practical point of view. For instance, if a system is described by a theory (i.e., by a set of formulas), expert may use the generator to get a quick insight into statements (represented by formulas) which follow from the theory. If the theory seems to be too general/restricted one can replace the theory by a new one (or modify the deductive system), run the generator again and observe how the set of proved formulas has changed. The process can be repeated until finally one obtains a desirable description of the system. The main advantage of the generator here is that (i) the user need not check large amount of statements by hand (ii) the user gets with each proved formula its proof written in a convenient way, thus, if the user does need to look at the proof, it is available. Note that the generator has been used this way during the exploration of basic properties of Galois connection with hedges. Another application of the prover may be an experimental determination of critical formulas, i.e. formulas which are (from some point of view) hard to prove. This has been partially demonstrated in Section IV-B.

Database of proofs from this paper can be found at:
`http://vychodil.inf.upol.cz/res/devel/aureas/`

Future research will focus on:

- improvements of the method,
  (i.e., merging of proofs which are build simultaneously)
- connections to evolutionary techniques,
  (genetic algorithms, genetic programming, ...)
- generating of finite models,
  (an automated method for generating of counterexamples)
- further experiments.

Acknowledgment

Supported by grant #1ET101370417 of the Grant Agency of the Academy of Sciences of Czech Republic and by institutional support, research plan MSM 6198959214.

References

[1] Bělohlávek R., Funioková T., Vychodil V: Galois connections with truth stressers: foundations for formal concept analysis of object-attribute data with fuzzy attributes. In: Reusch B. (Ed.): *Computational Intelligence, Theory and Applications*. Series: Advances in Soft Computing, pp. 205–219. Springer-Verlag, Berlin Heidelberg, 2005.
[2] Bělohlávek R., Funioková T., Vychodil V: Galois connections with hedges. In: *Proc. 11th IFSA Congress*, pp. 1250–1255 (vol. II), Tsinghua Univ. Press and Springer, 2005.
[3] Bělohlávek R., Vychodil V.: Fuzzy attribute logic: attribute implications, their validity, entailment, and non-redundant basis. In: *Proc. 11th IFSA Congress*, pp. 622–627 (vol. I), Tsinghua Univ. Press and Springer, 2005.
[4] Birkhoff G.: On the structure of abstract algebras. *Proc. Camb. Philos. Soc.* **31**(1935), 433–454.
[5] Burris S., Sankappanavar H. P.: *A Course in Universal Algebra*. Springer-Verlag, New York, 1981.
[6] Hájek P.: *Metamathematics of Fuzzy Logic*. Kluwer, Dordrecht, 1998.
[7] Hájek P.: On very true. *Fuzzy Sets and Systems* **124**(2001), 329–333.
[8] Klop J. W.: Term rewriting systems: A tutorial. *Bulletin EATCS* **32**, 143–182, 1987.
[9] Koza J. R.: *Genetic Programming*. MIT Press, Cambridge, MA, 1992.
[10] Knuth D. F., Bendix P. B.: Simple word problems in universal algebra. In: Leech, J. (Ed.): *Computational Problems in Abstract Algebra*. Oxford: Pergamon Press 1970, pp. 263–297.
[11] Lloyd, J. W.: *Foundations of Logic Programming* (2nd ed.). Springer-Verlag, New York, 1987.
[12] Mendelson E.: *Introduction to Mathematical Logic* (4th ed.). Chapman & Hall, London, 1997.
[13] Pavelka J.: On fuzzy logic I, II, III. *Z. Math. Logik Grundlagen Math.* **25**(1979), 45–52, 119–134, 447–464.
[14] Wechler W.: *Universal Algebra for Computer Scientists*. Springer-Verlag, Berlin Heidelberg, 1992.