Reusing Proofs

Thomas Kolbe and **Christoph Walther**

Abstract.[1] We develop a learning component for a theorem prover designed for verifying statements by mathematical induction. If the prover has found a proof, it is analyzed yielding a so-called *catch*. The catch provides the features of the proof which are relevant for reusing it in subsequent verification tasks and may also suggest useful lemmata. Proof analysis techniques for computing the catch are presented. A catch is generalized in a certain sense for increasing the reusability of proofs. We discuss problems arising when learning from proofs and illustrate our method by several examples.

1 INTRODUCTION

The improvement of problem solvers by reusing previously computed solutions is an active research area of Artificial Intelligence, emerging in the methodologies of *explanation-based learning* (EBL) [11, 4, 5] and *analogical reasoning* (AR) [2, 7, 12]. In EBL a problem's solution is analyzed, yielding an explanation why the solution succeeds. After generalization, the explanation is used for solving (similar) new problems. In AR a problem's solution guides the solution of (similar) new problems by suggesting corresponding inference steps. We present an approach for reusing *proofs* that combines ideas of EBL and AR as well as ideas from *abstraction* techniques [13, 6].

The domain of a problem solver must exhibit a certain regularity, i.e. solutions have to be similar in some sense, because otherwise nothing can be learned for subsequent problem solving. Of course, these handwaving notions of "regularity" and "similarity" must be made explicit for a specific problem solver before a learning component can be developed.

Theorem proving by (mathematical) induction is an area of automated reasoning where proofs often are similar and can be obtained in a uniform way, see e.g. [14] for a survey.[2] For proving some statement φ an induction theorem prover computes a set $\{\varphi_0, \ldots, \varphi_n\}$ of *induction formulas* for φ such that $[\varphi_0 \wedge \ldots \wedge \varphi_n \rightarrow \varphi]$ is an *induction axiom*. Then the system tries to infer each induction formula φ_i from the set of axioms AX by first-order means,[3] i.e. it is tested whether $AX \vdash \varphi_i$. Each induction formula φ_i has the form

$$(1.1) \quad \forall x^* \ (\forall u_1^* \ H_1 \wedge \ldots \wedge \forall u_n^* \ H_n) \ \rightarrow C$$

where C is the *induction conclusion* and each formula $\forall u_j^* \ H_j$ is an *induction hypothesis* (* denotes a sequence operator). If φ_i is a *step formula* at least one induction hypothesis exists, and (1.1) degenerates to $\forall x^* \ C$ in case of a *base formula*.

When an induction formula is proved, the induction conclusion is modified by successive applications of axioms and induction hypotheses until a tautology is derived. This creates a severe search problem because it frequently must be decided *when* and *where which* axiom or induction hypothesis should be applied such that a tautology eventually is obtained. This problem is one of the main challenges in automated mathematical induction and therefore studied intensively, cf. [3, 9, 14].

The work presented here is based on the following scenario: We assume an automated theorem prover which shall be supplemented by a learning component. Once the prover has computed an induction proof, the proof is *analyzed* and then *generalized* in a certain sense such that it can be *reused* subsequently. If the system is asked to prove another statement, it first looks for a previously computed proof of a *similar* statement and tries to reuse it. If the reuse fails the prover has to compute an original proof for the new statement. But if the proof reuse is successful no search for applying the right axioms and induction hypotheses is required, and this is what is saved.

2 REUSING PROOFS — AN EXAMPLE

We illustrate our method by an example. Let the functions plus, sum and app be defined by the following axioms where 0 and $\mathsf{s}(x)$ (resp. empty and $\mathsf{add}(n,x)$) are the constructors of the sort number (resp. list):[4]

$$\begin{array}{lll}
(\mathsf{plus\text{-}1}) & \mathsf{plus}(0,y) & = & y \\
(\mathsf{plus\text{-}2}) & \mathsf{plus}(\mathsf{s}(x),y) & = & \mathsf{s}(\mathsf{plus}(x,y)) \\
(\mathsf{sum\text{-}1}) & \mathsf{sum}(\mathsf{empty}) & = & 0 \\
(\mathsf{sum\text{-}2}) & \mathsf{sum}(\mathsf{add}(n,x)) & = & \mathsf{plus}(n,\mathsf{sum}(x)) \\
(\mathsf{app\text{-}1}) & \mathsf{app}(\mathsf{empty},y) & = & y \\
(\mathsf{app\text{-}2}) & \mathsf{app}(\mathsf{add}(n,x),y) & = & \mathsf{add}(n,\mathsf{app}(x,y))
\end{array}$$

These defining equations form a theory which may be extended by *lemmata*, i.e. statements which can be (inductively) infered from the defining equations and other already proved statements. For instance

---

[1] Fachbereich Informatik, Technische Hochschule Darmstadt, Alexanderstr. 10, D-64283 Darmstadt, Germany. Email: {kolbe,walther}@inferenzsysteme.informatik.th-darmstadt.de
This work was supported under grants no. Wa652/4-1,2 by the Deutsche Forschungsgemeinschaft as part of the focus program "Deduktion".

[2] Throughout this paper *induction* stands for *mathematical induction* and should not be confused with induction in the sense of machine learning.

[3] Induction proofs may be nested, i.e. a further induction may be required to prove φ_i. But for the sake of the presentation we consider only non-nested inductions here.

[4] We usually omit universal quantifiers at the top level of formulas as well as the sort information for variables.

ECAI 94. *11th European Conference on Artificial Intelligence* Edited by A. Cohn
Published in 1994 by John Wiley & Sons, Ltd.

$$\text{(lem-1)}\quad \mathsf{plus}(\mathsf{plus}(x,y),z) = \mathsf{plus}(x,\mathsf{plus}(y,z))$$

can be easily proved and therefore may be used like any defining equation in subsequent deductions. We aim to optimize proving such lemmata by reusing previously computed proofs of other lemmata. For instance consider the statement

$$\varphi_1[x,y] :\equiv \ \mathsf{plus}(\mathsf{sum}(x),\mathsf{sum}(y)) = \mathsf{sum}(\mathsf{app}(x,y)).$$

To prove the conjecture $\forall x,y\ \varphi_1[x,y]$ two induction formulas, viz. the base formula φ_{1b} and the step formula φ_{1s} are generated

$$\begin{array}{lcl} \varphi_{1b} & :\equiv & \forall y\ \varphi_1[\mathsf{empty},y] \\ \varphi_{1s} & :\equiv & \forall n,x,y\ (\forall u\ \varphi_1[x,u]) \ \rightarrow \varphi_1[\mathsf{add}(n,x),y]. \end{array}$$

The following proof of the step formula φ_{1s} is obtained by modifying the induction conclusion $\varphi_1[\mathsf{add}(n,x),y] \equiv$

$$\mathsf{plus}(\mathsf{sum}(\mathsf{add}(n,x)),\mathsf{sum}(y)) = \mathsf{sum}(\mathsf{app}(\mathsf{add}(n,x),y)) \qquad \text{IC}$$

in a backward chaining style, i.e. each statement is implied by the statement in the line below, where terms are underlined if they have been changed:[5]

$$\begin{array}{lcl} \mathsf{plus}(\mathsf{sum}(\mathsf{add}(n,x)),\mathsf{sum}(y)) = & \ldots & \text{IC} \\ \mathsf{plus}(\underline{\mathsf{plus}(n,\mathsf{sum}(x))},\mathsf{sum}(y)) = & \ldots & \text{(sum-2)} \\ \quad\ldots & = \mathsf{sum}(\underline{\mathsf{add}(n,\mathsf{app}(x,y))}) & \text{(app-2)} \\ \quad\ldots & = \underline{\mathsf{plus}(n,\mathsf{sum}(\mathsf{app}(x,y)))} & \text{(sum-2)} \\ \quad\ldots & = \mathsf{plus}(n,\underline{\mathsf{plus}(\mathsf{sum}(x),\mathsf{sum}(y))}) & \text{IH} \\ \underline{\mathsf{plus}(n,\mathsf{plus}(\mathsf{sum}(x),\mathsf{sum}(y)))} = & \ldots & \text{(lem-1)} \\ \underline{\mathsf{true}} & & x = x \end{array}$$

Having computed a proof, it is *analyzed* to distinguish its *relevant* features from its *irrelevant* parts. Relevant features are specific to the proof and are collected in a so-called *proof catch* because "similar" requirements must be satisfied if this proof is to be reused later on. We consider features like the positions where equations are applied, induction conclusions and hypotheses, general laws as reflexivity etc. as irrelevant because they can always be satisfied.

Analysis of the above proof yields (sum-2), (app-2) and (lem-1) as the catch. E.g. all we have to know about plus is its associativity, but not its semantics or how plus is computed. We then *generalize* the conjecture, the induction formulas and the catch for obtaining a so-called *proof shell*. This is achieved by replacing function *symbols* by function *variables* denoted by capital letters F,G,H etc. yielding the *schematic conjecture* $\Phi_1[F,G,H][x,y]$ with the corresponding *schematic induction formulas* Φ_{1b} and Φ_{1s} as well as the *schematic catches* Φ'_{1b} and Φ'_{1s}:

$$\begin{array}{lcl} \Phi_1[F,G,H][x,y] & :\equiv & F(G(x),G(y)) = G(H(x,y)) \\ \Phi_{1b}[F,G,H][C] & :\equiv & \forall y\ \Phi_1[F,G,H][C,y] \\ \Phi_{1s}[F,G,H][D] & :\equiv & \forall n,x,y\ (\forall u\ \Phi_1[F,G,H][x,u]) \\ & & \qquad \rightarrow \Phi_1[F,G,H][D(n,x),y] \\ \Phi'_{1b}[F,G,H][C] & :\equiv & \ldots \\ & & (2.1)\quad G(D(n,x)) \quad = F(n,G(x)) \\ \Phi'_{1s}[F,G,H][D] & :\equiv & (2.2)\quad H(D(n,x),y) = D(n,H(x,y)) \\ & & (2.3)\quad F(F(x,y),z) \quad = F(x,F(y,z)) \end{array}$$

Figure 1. The proof shell PS_1 for the proof of φ_1

If a new statement ψ shall be proved, a set of induction formulas I_ψ is computed for ψ. Then it is tested whether some proof shell PS exists which *applies for* ψ, i.e. whether (i) ψ is an instance of the schematic conjecture of PS and (ii) I_ψ is an instance of the set of schematic induction formulas of PS. If both tests succeed, the matcher obtained from the second test is applied to the schematic catches of PS. If the formulas of all instantiated schematic catches can be proved (which may necessitate further proof reuses), ψ is verified by reuse since the truth of an instantiated schematic catch implies the truth of its instantiated schematic induction formula, cf. [10].[6]

E.g. assume that the new conjecture $\forall x,y\ \psi_1[x,y]$ shall be proved, where

$$\psi_1[x,y] :\equiv \ \mathsf{times}(\mathsf{prod}(x),\mathsf{prod}(y)) = \mathsf{prod}(\mathsf{app}(x,y))$$

and times and prod are defined by the axioms

$$\begin{array}{lll} \text{(times-1)} & \mathsf{times}(0,y) & = \ 0 \\ \text{(times-2)} & \mathsf{times}(\mathsf{s}(x),y) & = \ \mathsf{plus}(y,\mathsf{times}(x,y)) \\ \text{(prod-1)} & \mathsf{prod}(\mathsf{empty}) & = \ \mathsf{s}(0) \\ \text{(prod-2)} & \mathsf{prod}(\mathsf{add}(n,x)) & = \ \mathsf{times}(n,\mathsf{prod}(x)). \end{array}$$

The induction formulas computed for ψ_1 are

$$\begin{array}{lcl} \psi_{1b} & :\equiv & \forall y\ \psi_1[\mathsf{empty},y] \\ \psi_{1s} & :\equiv & \forall n,x,y\ (\forall u\ \psi_1[x,u]) \ \rightarrow \psi_1[\mathsf{add}(n,x),y]. \end{array}$$

Obviously ψ_1 is an instance of Φ_1 and $\{\psi_{1b},\psi_{1s}\}$ is an instance of $\{\Phi_{1b},\Phi_{1s}\}$. Hence (only considering the step case) we may try to reuse the given proof by instantiating the schematic catch Φ'_{1s} and subsequent verification of the resulting proof obligations:

$$\begin{array}{llcl} \multicolumn{4}{l}{\Phi'_{1s}[\mathsf{times},\mathsf{prod},\mathsf{app}][\mathsf{add}] \ \equiv} \\ (2.4) & \mathsf{prod}(\mathsf{add}(n,x)) & = & \mathsf{times}(n,\mathsf{prod}(x)) \\ (2.5) & \mathsf{app}(\mathsf{add}(n,x),y) & = & \mathsf{add}(n,\mathsf{app}(x,y)) \\ (2.6) & \mathsf{times}(\mathsf{times}(x,y),z) & = & \mathsf{times}(x,\mathsf{times}(y,z)) \end{array}$$

Features (2.4) and (2.5) are axioms, viz. (prod-2) and (app-2), so it only remains to prove the associativity of times (i.e. a required lemma generated in a *goal directed* way) and, if successful, ψ_{1s} is proved. Compared to a direct proof of ψ_{1s} we have no search control problems, as the associativity of times must be verified in either case. This motivates why we expect our method to improve the efficiency of the theorem prover.

3 SIMPLE PROOF ANALYSIS

To formalize the so-called *simple proof analysis* which was illustrated in the previous section, we start by defining the calculus in which we prove our (base and step) formulas. Each rule of this calculus is built from a "conventional" inference rule by stipulating in addition which formula has to be remembered as a "relevant feature" if the particular inference rule is used in a proof. Hence the rules are applied to expressions of the form $\langle\varphi, A\rangle$, where φ is a formula and A, called the *accumulator*, holds the catch collected so far.

[5] We omit a proof for the base formula φ_{1b} here and in subsequent examples as there are no particularities compared to the step case.

[6] Since requirement (ii) implies requirement (i), the first test can be abandoned without loosing the soundness of the approach. But we insist on the first test for economical reasons, because it is much more expensive to test for requirement (ii) than to test for requirement (i). So if the cheap test (i) fails, no further effort must be spent for the reuse attempt.

Since we want to investigate the problems of proof analysis and reuse, we confine ourselves here with unconditional equations as the only conjectures to be proved. This eases the presentation but does not restrict the applicability of our results as the extension to general first-order formulas is straightforward (for the price of additional formal clutter and further inference rules, of course [10]). However when proving a conjecture we consider also induction formulas of the form (1.1).

Let AX denote the set of all axioms (i.e. defining equations and lemmata) from which inferences are drawn. For terms t, s and an occurrence $p \in \mathrm{Occ}(t)$ the subterm of t at occurrence p is denoted by $t|_p$ and $t[p \leftarrow s]$ is the term obtained by replacing $t|_p$ in t by s. $\mathcal{V}(t)$ denotes the set of all variables in t.

Our calculus consists of three rules, viz. a logical axiom and two restricted versions of paramodulation, where Γ denotes the set of induction hypotheses, σ denotes a substitution and $\mathrm{dom}(\sigma)$ denotes its domain:

$$\text{Reflexivity} \quad \frac{\langle \forall x^* \ \Gamma \rightarrow t = t, \ A \rangle}{\langle \mathsf{true}, \ A \rangle}$$

$$AX\text{-replacement} \quad \frac{\langle \forall x^* \ \Gamma \rightarrow C, \ A \rangle}{\langle \forall x^* \ \Gamma \rightarrow C[p \leftarrow \sigma(r)], \ A \cup \{\forall u^* \ l = r\} \rangle}$$
$$\text{if } \ [\forall u^* \ l = r] \in AX, \ C|_p = \sigma(l) \text{ and } \mathcal{V}(r) \subseteq \mathcal{V}(l) \subseteq u^*$$

$$HYP\text{-replacement} \quad \frac{\langle \forall x^* \ \Gamma \rightarrow C, \ A \rangle}{\langle \forall x^* \ \Gamma \rightarrow C[p \leftarrow \sigma(r)], \ A \rangle}$$
$$\text{if } \ [\forall u^* \ l = r] \in \Gamma, \ C|_p = \sigma(l), \ \mathcal{V}(r) \subseteq \mathcal{V}(l), \ \mathrm{dom}(\sigma) \subseteq u^*$$

The replacement rules differ only in updating the accumulator component: If the applied equation $[\forall u^* \ l = r]$ is an *axiom* it has to be recorded in A for obtaining the catch, but if the applied equation is an induction *hypothesis* it is irrelevant for the learning step and A remains unchanged.

Example 1 Consider the expression $\langle \forall ... \ \Gamma \rightarrow g(f(c, h(y))) = ..., \ \{...\} \rangle$ and the equation $E :\equiv [\forall x \ f(c, x) = h(x)]$. If $E \in \Gamma$, then the expression $\langle \forall ... \ \Gamma \rightarrow g(h(h(y))) = ..., \ \{...\} \rangle$ is obtained by HYP-replacement. But if $E \in AX$, then AX-replacement yields $\langle \forall ... \ \Gamma \rightarrow g(h(h(y))) = ..., \ \{E, ...\} \rangle$.

If $\langle \forall x^* \ \Gamma \rightarrow C, \ \emptyset \rangle \vdash \langle \mathsf{true}, \ A \rangle$ can be established, then $[\forall x^* \ \Gamma \rightarrow C]$ is proved and A contains the catch of the proof. Now given a formula φ and a set of induction formulas $\{\varphi_0, \ldots, \varphi_n\}$ for φ, we try to infer $\langle \varphi_i, \ \emptyset \rangle \vdash \langle \mathsf{true}, \ A_i \rangle$ for each i in our calculus. If successful, φ is proved and for each i an accumulator A_i is obtained which holds the catch of the proof of φ_i. Then we replace each function symbol s occurring in $\varphi, \varphi_0, \ldots, \varphi_n, A_0, \ldots, A_n$ by a function variable S yielding a proof shell with the schematic conjecture Φ, the schematic induction formulas $\Phi_0, \ldots, \Phi_n$ and the schematic catches $\Phi'_0, \ldots, \Phi'_n$.

4 REFINED PROOF ANALYSIS

The success of proof reuses directly depends on the generality of what has been learned from a given proof. Consider the statement

$$\psi_2[x, y] :\equiv \ \mathsf{plus}(\mathsf{len}(x), \mathsf{len}(y)) = \mathsf{len}(\mathsf{app}(x, y))$$

where the length $\mathsf{len}(x)$ of a list x is defined by

$$\begin{array}{lll} (\mathsf{len}\text{-}1) & \mathsf{len}(\mathsf{empty}) & = \ 0 \\ (\mathsf{len}\text{-}2) & \mathsf{len}(\mathsf{add}(n, x)) & = \ \mathsf{s}(\mathsf{len}(x)). \end{array}$$

Since PS_1 (cf. Fig. 1) applies for ψ_2, we instantiate the schematic catch Φ'_{1s} (only considering the step case) yielding the proof obligations

$$\begin{array}{llll} (4.1) & \mathsf{len}(\mathsf{add}(n, x)) & = & \mathsf{plus}(n, \mathsf{len}(x)) \\ (4.2) & \mathsf{app}(\mathsf{add}(n, x), y) & = & \mathsf{add}(n, \mathsf{app}(x, y)) \\ (4.3) & \mathsf{plus}(\mathsf{plus}(x, y), z) & = & \mathsf{plus}(x, \mathsf{plus}(y, z)). \end{array}$$

But statement (4.1) does not hold, hence ψ_2 cannot be verified by reuse. One idea is that ψ_2 requires an original proof which differs in its structure from all previously computed proofs, and therefore the proof reuse fails. But it may also be the case, that what has been learned is simply not enough, so the reuse fails only for this reason.

The latter is true for statement ψ_2 and we improve our technique so that the reuse eventually is successful. The key idea for the improvement is to distinguish different *occurrences* of function symbols in the conjecture and in the catch of a proof. For instance statement φ_1 from section 2 will now be generalized to $\Phi_2[F^1, G^1, G^2, G^3, H^1][x, y] :\equiv$

$$F^1(G^1(x), G^2(y)) = G^3(H^1(x, y))$$

with function variables F^1, G^1, G^2, G^3, H^1 (instead of generalizing it to $\Phi_1[F, G, H][x, y]$ as before). This necessitates modifications of the proof analysis yielding the technique of *refined proof analysis*.

To distinguish the different occurrences of function symbols we label function symbols with an *index*: For a signature Σ let $\Sigma^{\mathbb{N}} := \{f^i \mid f \in \Sigma, \ i \in \mathbb{N}\}$ be the corresponding *index-signature*. Then $\mathcal{T}(\Sigma^{\mathbb{N}}, \mathcal{V})$ are the *index-terms* (*i-terms* for short) built from $\Sigma^{\mathbb{N}}$ and $\mathcal{V}$. Let $index : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma^{\mathbb{N}}, \mathcal{V})$ be a mapping that supplies all occurrences of all function symbols in a term with new unique indices,[7] e.g. $index(\mathsf{s}(\mathsf{plus}(\mathsf{s}(x), y))) = \mathsf{s}^1(\mathsf{plus}^1(\mathsf{s}^2(x), y))$. We demand *index* to yield indices in ascending order starting with 1.

We call a subset $K \subseteq \{\langle f^i, f^j \rangle \mid f \in \Sigma, \ i, j \in \mathbb{N}\}$ of $\Sigma^{\mathbb{N}} \times \Sigma^{\mathbb{N}}$ an *(index-) collision set* and $\sim_K$ denotes the reflexive, transitive and symmetric closure of K. Hence $\sim_K$ is an equivalence relation which can be extended to i-terms yielding a congruence relation $\sim_K$ on $\mathcal{T}(\Sigma^{\mathbb{N}}, \mathcal{V})$, i.e. an equivalence relation such that $t_1 \sim_K s_1, \ldots, t_n \sim_K s_n$ and $f^i \sim_K f^j$ implies $f^i(t_1, \ldots, t_n) \sim_K f^j(s_1, \ldots, s_n)$. We write $t =_K s$ iff K is a *minimal* collision set satisfying $t \sim_K s$. Hence $t =_K s$ implies $t \sim_K s$ but not vice versa.

Matching is extended to i-terms by ignoring the indices when the clash test is performed. This means that e.g. $\{x/a^1\}$ is a matcher of $f^1(x)$ and $f^2(a^1)$ whereas $f^1(x)$ and $g^1(a^1)$ are non-matchable i-terms. So if σ is a matcher of the i-terms t and s, then $\sigma(t) =_K s$ for some K.

We modify our calculus from section 3 to incorporate the bookkeeping of the indices. The modified calculus operates on triples $\langle \forall x^* \ \Gamma \rightarrow C, \ K, \ A \rangle$, where all terms in Γ, C and A are i-terms now and an additional component, viz. the collision

[7] Strictly speaking, *index* is an operation having a side effect since we demand that indices obtained by *index* have been "never used before".

set K, keeps track of the function symbols from $\Sigma^{\mathbb{N}}$ which have been identified in a proof, but differ in their indices:

$$\text{Reflexivity} \quad \frac{\langle \forall x^* \; \Gamma \rightarrow s = t, \; K, \; A\rangle}{\langle \mathsf{true}, \; K \cup K', \; A\rangle} \qquad \text{if} \;\; s =_{K'} t$$

AX-replacement

$$\frac{\langle \forall x^* \; \Gamma \rightarrow C, \; K, \; A\rangle}{\langle \forall x^* \; \Gamma \rightarrow C[p \leftarrow \sigma(r')], \; K \cup K', \; A \cup \{\forall u^* \; l' = r'\}\rangle}$$

$$\text{if} \;\; [\forall u^* \; l = r] \in AX, \; l' := index(l), \; r' := index(r), \; C|_p =_{K'} \sigma(l') \text{ and } \mathcal{V}(r) \subseteq \mathcal{V}(l) \subseteq u^*$$

$$HYP\text{-replacement} \quad \frac{\langle \forall x^* \; \Gamma \rightarrow C, \; K, \; A\rangle}{\langle \forall x^* \; \Gamma \rightarrow C[p \leftarrow \sigma(r)], \; K \cup K', \; A\rangle}$$

$$\text{if} \;\; [\forall u^* \; l = r] \in \Gamma, \; C|_p =_{K'} \sigma(l), \; \mathcal{V}(r) \subseteq \mathcal{V}(l), \; \mathrm{dom}(\sigma) \subseteq u^*$$

Both replacement rules not only differ in updating the accumulator component (as they already do in the simple analysis approach), but they differ also in the treatment of the function indices: AX-replacement generates a freshly indexed variant of an axiom before application whereas HYP-replacement applies an already indexed hypothesis without index modifications. Both rules record in the collision component the indexed function symbols which have to be identified in the replacement step.

Example 2 We resume Example 1 from section 3. Consider the indexed expression $\langle \forall ... \; \Gamma \rightarrow g^2(f^2(c^2, h^2(y))) = ..., \{...\}, \{...\}\rangle$ and the equation $E :\equiv [\forall x \; f(c, x) = h(x)]$. If (an indexed version of) E is in Γ, e.g. $[\forall x \; f^1(c^1, x) = h^1(x)] \in \Gamma$, then $\langle \forall ... \; \Gamma \rightarrow g^2(h^1(h^2(y))) = ..., \; \{\langle f^2, f^1\rangle, \langle c^2, c^1\rangle, \; ...\}, \{...\}\rangle$ is obtained by HYP-replacement. But if $E \in AX$, then AX-replacement uses $index(E) :\equiv [\forall x \; f^3(c^3, x) = h^3(x)]$ yielding $\langle \forall ... \; \Gamma \rightarrow g^2(h^3(h^2(y))) = ..., \; \{\langle f^2, f^3\rangle, \langle c^2, c^3\rangle, \; ...\}, \; \{\forall x \; f^3(c^3, x) = h^3(x), \; ...\}\rangle$.

Note that we demand "$... =_{K'} ...$" instead of "$... \sim_{K'} ...$" in the definition of the inference rules. This entails that only function symbols are identified which *must* be identified and therefore guarantees that a *most general* (schematic) catch will be obtained subsequently. Otherwise e.g. the pair $\langle h^1, h^2\rangle$ in Example 2 could also be inserted into the collision set yielding an identification which is not required by the inference step.

In order to prove an induction formula φ_i we try to establish $\langle index(\varphi_i), \; \emptyset, \; \emptyset\rangle \vdash \langle \mathsf{true}, \; K, \; A\rangle$ in our calculus. The collision set K contains pairs $\langle f^i, f^j\rangle$ of indexed function symbols which have been identified in the proof. This information must be propagated into the accumulator A when building the proof catch, because the proof is based on the assumption that f^i and f^j denote *identical* functions: Either (i) f^i and f^j are identified *syntactically* by replacing f^i with f^j (or vice versa) in the equations of A or (ii) f^i and f^j are identified *semantically* by insertion of $[\forall x_1, ..., x_n \; f^i(x_1, ..., x_n) = f^j(x_1, ..., x_n)]$ into A.

The decision which kind of identification is used influences the generality of the proof shell and the effort which must be spent for reusing the proof. If only semantical identification is used, the proof shell is as general as possible but the schematic catch may provide a large number of proof obligations, thus increasing the effort for subsequent reuses. This effort is minimized if only syntactical identification is used. But then the applicability of the proof shell is decreased. Consequently, some compromise must be made between both extremes which is based on the evaluation of several examples, cf. [10].

Here we use a quite simple criterion for identification which is enough for the examples in this paper: For a *free* function symbol f^i from A (i.e. one that does not occur in the proved induction formula) let f^j be the function symbol from the $\sim_K$-equivalence class of f^i with the *smallest* index. Now replace f^i by f^j in A, i.e. use syntactical identification. *Bound* function symbols (i.e. those occurring in the proved induction formula) remain unchanged in A, but if $f^i \sim_K f^j$ for two bound function symbols $f^i \neq f^j$, then insert $[\forall x_1, ..., x_n \; f^i(x_1, ..., x_n) = f^j(x_1, ..., x_n)]$ into A, i.e. use semantical identification.

Since we assign indices in *ascending* order starting with the conjecture, it is guaranteed that a free function symbol f^i is always replaced by a bound function symbol if the $\sim_K$-equivalence class of f^i contains one bound function symbol at least. Thus the function symbols from the conjecture are propagated into the proof catch as far as possible (and necessary).

Now the equations in the modified accumulator A are generalized as before yielding the schematic catch, where however function symbols with *different indices* are replaced by *different function variables.*

We resume the proof of φ_1 from section 2. After assigning the indices we prove the induction formulas for the conjecture

$$\mathsf{plus}^1(\mathsf{sum}^1(x), \mathsf{sum}^2(y)) = \mathsf{sum}^3(\mathsf{app}^1(x, y))$$

and obtain a proof shell PS_2 with the schematic conjecture Φ_2 and the following schematic catch for the step formula:[8]

$$\begin{array}{llcl} (4.4) & G^1(D^1(n,x)) & = & F^2(n, G^1(x)) \\ (4.5) & H^1(D^1(n,x), y) & = & D^4(n, H^1(x,y)) \\ (4.6) & G^3(D^4(n,x)) & = & F^3(n, G^3(x)) \\ (4.7) & F^1(F^2(x,y), z) & = & F^3(x, F^1(y,z)) \end{array}$$

Now reconsider statement ψ_2 from the beginning of this section. Since PS_2 applies for ψ_2, the instantiated catch

$$\begin{array}{llcl} (4.8) & \mathsf{len}(\mathsf{add}(n,x)) & = & F^2(n, \mathsf{len}(x)) \\ (4.9) & \mathsf{app}(\mathsf{add}(n,x), y) & = & D^4(n, \mathsf{app}(x,y)) \\ (4.10) & \mathsf{len}(D^4(n,x)) & = & F^3(n, \mathsf{len}(x)) \\ (4.11) & \mathsf{plus}(F^2(x,y), z) & = & F^3(x, \mathsf{plus}(y,z)) \end{array}$$

is computed. This schematic catch is only *partially* instantiated because the function variables F^2, F^3 and D^4 stemming from the function symbols plus, plus and add in the catch of the original proof are not replaced by concrete function symbols. This is because these function variables do not occur in the schematic induction formula of the proof shell. We call such function variables of a proof shell *free* and we call all function variables occurring in the schematic induction formulas *bound* function variables.

[8] Dropping the indices in these equations yields the schematic catch Φ'_{1_s} from Fig. 1.

A formula φ with a free function variable F is *true* iff some function exists such that the formula obtained from φ by replacing F with this function can be proved. Thus e.g. a provable formula, viz. the axiom (app-2), is obtained from (4.9) if D^4 is replaced by add. However we are not restricted to function *symbols* but may use any (unnamed) *function* represented e.g. as an expression of the λ-calculus for the replacement of function variables. For instance, provable formulas are obtained from (4.8), (4.10) and (4.11) if F^2 and F^3 are replaced by $\lambda u, v.\mathsf{s}(v)$ as e.g. (4.8) is instantiated to the axiom

$$\text{(4.12)} \quad \mathsf{len}(\mathsf{add}(n,x)) = \mathsf{s}(\mathsf{len}(x))$$

and ψ_2 is proved by reuse only.

Formally, solutions for free function variables are computed by an algorithm for second-order matching "modulo evaluation" [10] which is based on the matching algorithm of *Huet* and *Lang* [8]. Note that the heart of success is the *free* function variable F^2 for which the right solution can be computed by this second-order matching. In the simple analysis approach, the *bound* function variable F stands in the place of F^2 (cf. Fig. 1). Consequently, F is instantiated with the function symbol plus from the conjecture thus falsifying equation (4.8).

Since also more general *schematic conjectures* are obtained when using the refined proof analysis, proofs can more often be reused only because a proof shell applies more often. Consider the associativity of multiplication and the statement $z^{xy} = (z^y)^x$, i.e.

$$\begin{aligned} \varphi_3[x,y,z] :\equiv \; & \mathsf{times}(\mathsf{times}(x,y),z) &= \mathsf{times}(x,\mathsf{times}(y,z)) \\ \psi_3[x,y,z] :\equiv \; & \mathsf{exp}(\mathsf{times}(x,y),z) &= \mathsf{exp}(x,\mathsf{exp}(y,z)) \end{aligned}$$

and suppose that φ_3 is already proved. With the *simple* proof analysis, the proof of φ_3 cannot be reused for verifying ψ_3 because ψ_3 is not an instance of the schematic conjecture

$$\Phi_3[F][x,y,z] :\equiv F(F(x,y),z) = F(x,F(y,z))$$

obtained from φ_3. But with the *refined* proof analysis, φ_3 is generalized to the schematic conjecture

$$\Phi_4[F^1,F^2,F^3,F^4][x,y,z] :\equiv F^1(F^2(x,y),z) = F^3(x,F^4(y,z))$$

which matches ψ_3, so now the proof of φ_3 can be reused, cf. [10]. Here the occurrences of times in φ_3 correspond to the *distinct* function symbols exp and times in ψ_3, i.e. the artificially generalized schematic catch generated from the proof of φ_3 is sufficient to prove the *more general* conjecture ψ_3.

A further increase of reusability is obtained by miscellaneous modifications of the matching algorithm concerning the order and presence of arguments in function applications. Thus e.g.

$$\begin{aligned} &\mathsf{app}(\mathsf{reverse}(y),\mathsf{reverse}(x)) &&= \mathsf{reverse}(\mathsf{app}(x,y)) \\ &\mathsf{app}(\mathsf{remove}(n,x),\mathsf{remove}(n,y)) &&= \mathsf{remove}(n,\mathsf{app}(x,y)) \\ &\mathsf{plus}(\mathsf{double}(x),\mathsf{double}(y)) &&= \mathsf{double}(\mathsf{plus}(x,y)) \end{aligned}$$

all can be verified by reusing the proof of φ_1, cf. [10].

5 CONCLUSION

Our method for reusing proofs is related to *abstraction techniques* in problem solving [13, 6] as well as to the machine learning methodologies of *explanation-based learning* [11, 4, 5] and *reasoning by analogy* [2, 7, 12], see [10] for a comparison with our method. The usefulness of our proposal depends on the frequency and the costs of proof reuses in realistic applications, and therefore can only be evaluated after experiments of appropriate size have been carried out.

We have just finished the implementation of a system called PLAGIATOR [1]. This system consists of a device for analyzing, generalizing and managing proofs and is based on the techniques discussed above. If a statement cannot be verified by reuse, the user must support the system with a proof which then is analyzed etc. The creation of a hand crafted proof is supported by a proof editor which is also part of the system. The proof editor only checks whether inference rules are legally applied but offers no further support in finding the proof. This means that the system is really weak in its problem solving ability, thus motivating the system's name — the German word for plagiarist — as it is intended to obtain a system exhibiting an intelligent behavior only because it is able to adapt solutions provided by other intelligent devices.

The PLAGIATOR-system will be used in the next step of our research for developing heuristics for an efficient retrieval and application of learning results and subsequent evaluation of the proposal.

REFERENCES

[1] J. Brauburger. PLAGIATOR: Entwurf und Implementierung eines lernenden Beweisers. Diploma Thesis, TH Darmstadt, 1994.

[2] B. Brock, S. Cooper, and W. Pierce. Analogical Reasoning and Proof Discovery. In *Proceedings of the 9th International Conference on Automated Deduction,* Argonne, pages 454–468, 1988.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence*, 62:183–253, 1993.

[4] G. DeJong and R. Mooney. Explanation-based Learning: An Alternative View. *Machine Learning*, 1:145–176, 1986.

[5] T. Ellman. Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Surveys*, 21(2):163–221, 1989.

[6] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[7] R. P. Hall. Computational Approaches to Analogical Reasoning: A Comparative Analysis. *Artificial Intelligence*, 39:39–120, 1989.

[8] G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11:31–55, 1978.

[9] D. Hutter. Guiding Induction Proofs. In *Proceedings of the 10th International Conference on Automated Deduction,* Kaiserslautern, pages 147–161, 1990.

[10] T. Kolbe and C. Walther. Reusing Proofs — A first Report. Technical report, TH Darmstadt, 1994.

[11] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based Generalization: A Unifying View. *Machine Learning*, 1:47–80, 1986.

[12] S. Owen. *Analogy for Automated Reasoning.* Academic Press, 1990.

[13] D. A. Plaisted. Theorem Proving with Abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[14] C. Walther. Mathematical Induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 2. Oxford University Press, 1994.