Proving Theorems by Mimicking a Human's Skill

Thomas Kolbe Christoph Walther
Fachbereich Informatik, Technische Hochschule Darmstadt,
Alexanderstr. 10, D-64283 Darmstadt, Germany.
Email: {kolbe|walther}@inferenzsysteme.informatik.th-darmstadt.de

Abstract.[1] We investigate the improvement of theorem provers by reusing previously computed proofs. We have developed and implemented the PLAGIATOR system which proves theorems by mathematical induction with the aid of a human advisor: If a conjecture is submitted to the system, it tries to reuse a proof of a previously verified conjecture. If successful, resources are saved, because the number of required user interactions is decreased. The performance of the overall system is improved, because *necessary lemmata* might be *speculated*. If the reuse fails, the human advisor is called for providing a hand crafted proof for such a conjecture, which subsequently — after some (automated) preparation steps — is stored in the system's memory, to be in stock for future reasoning problems. The success of our approach is based on our technique for *preparing* given proofs as well as by our technique for *reusing* proofs.

Introduction

We investigate the improvement of theorem provers by reusing previously computed proofs, cf. (Kolbe & Walther 1994, Kolbe & Walther 1995b). Our work has similarities with the methodologies of *explanation-based learning* (Ellman 1989), *analogical reasoning* (Hall 1989), and *abstraction* (Giunchiglia & Walsh 1992, Villafiorita & Giunchiglia 1996).

Consider the following general architecture: Some problem solver PS is augmented with a facility for storing and retrieving solutions of problems solved during the system's lifetime. The problem solver either can be some machine, a machine supported interactively by a human advisor, or a human only. One can think of several benefits by providing some memory for making a problem solver cognizant of previous work:

1. the quality of the solution process is improved (i.e. less resources are required as compared to problem solving from scratch);
2. the performance of the problem solver is improved (i.e. more problems are solvable as compared to problem solving from scratch);
3. the quality of solutions is improved (e.g. a better plan, if PS is a planner).

The presence and the degree of these benefits strongly depend on the quality of the problem solver and the domain it is operating on. Here we consider a domain where *problems* are *conjectures* to be *proved*. We have developed and implemented the PLAGIATOR system (Brauburger 1994) which proves theorems by *mathematical induction*[2] in the spirit of the *problem reduction paradigm* (Nilsson 1971): If a conjecture is submitted to the system, it tries to find a proof by inspecting its memory (called a *proof dictionary*) for *reusing* proofs of previously verified conjectures. If successful, the retrieval results in a set of conjectures, the truth of which is sufficient for the truth of the given conjecture. Then for each of these retrieved conjectures, the proof dictionary is searched again for reusable proofs and so on, until eventually a retrieved conjecture either is obviously true or the retrieval fails. In the latter case, a human advisor is called for providing a hand crafted proof for such a conjecture, which subsequently — after some (automated) preparation steps — is stored in the proof dictionary to

[1] This work was supported under grants no. Wa652/4-1,2,3 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.

be in stock for future reasoning problems.

In this way the system shall exhibit an intelligent behavior, although it is unable to find an original proof by its own, thus motivating the system's name, viz. the German word for plagiarist. Our approach has two benefits, as several experiments with the PLAGIATOR system reveal (Kolbe & Walther 1995d): (1) Resources are saved, because the number of required user interactions is decreased. (2) The performance of the overall system is improved, because the PLAGIATOR system is able to *speculate lemmata*, which are helpful to prove a given conjecture. The latter feature is particularly important, because it is retained if the human advisor is substituted by a machine, i.e. an automated induction theorem prover, cf. (Bouhoula, Kounalis, & Rusinowitch 1992, Boyer & Moore 1979, Bundy *et al.* 1990, Kapur & Zhang 1988, Walther 1994): Many domains, such as induction theorem proving or planning, do not have *complete* problem solvers, i.e. problem solvers which solve each solvable problem. Then the speculation of useful subgoals yields a relevant improvement of the system's problem solving performance.

The success of our approach is based on our technique for *preparing* given proofs (by *proof analysis* and *generalization*) to store them into the proof dictionary, as well as by our technique for reusing proofs (by *retrieval* and *adaptation* methods). Subsequently a survey of our approach for reusing proofs is presented and its advantages are illustrated by examples.

Reusing Proofs — An Example

Let us briefly sketch our method for reusing proofs (see (Kolbe & Walther 1994) for more details): An *induction formula* $IH \rightarrow IC$ is either a *step* formula or a *base* formula in which case IH equals TRUE. Induction formulas are proved by modifying the induction conclusion IC using given axioms until the induction hypothesis IH is applicable.

For instance, 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):[3]

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

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

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

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

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

We prove the conjecture $\forall x,y\ \varphi[x,y]$ by induction upon the list-variable x and obtain two induction formulas, viz. the base formula φ_b and the step formula φ_s as

$$\begin{array}{lll}
\varphi_b & := & \forall y\ \varphi[\mathsf{empty},y] \\
\varphi_s & := & \forall n,x,y\ (\forall u\ \varphi[x,u]) \rightarrow \varphi[\mathsf{add}(n,x),y].
\end{array}$$

The following proof of the step formula φ_s is obtained by modifying the induction conclusion

$$\mathrm{IC} := \varphi[\mathsf{add}(n,x),y] = \mathsf{plus}(\mathsf{sum}(\mathsf{add}(n,x)),\mathsf{sum}(y)) \equiv \mathsf{sum}(\mathsf{app}(\mathsf{add}(n,x),y))$$

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 in the corresponding proof step:[4]

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

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

[4] We omit a proof for the base formula φ_b as there are no particularities compared to the step case.

Given such 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 *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 $x \equiv x$ etc. as irrelevant because they can always be satisfied. So the catch of a proof is a *subset* of the set of leaves of the corresponding proof tree.

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 for proving $\forall x, y\ \varphi[x, y]$ is its associativity, but not its semantics or *how* plus is computed. We then *generalize*[5] the conjecture, the induction formula 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* $\Phi[x, y] := F(G(x), G(y)) \equiv G(H(x, y))$ with the corresponding *schematic induction formula* Φ_s as well as the *schematic catch* C_s:

$$\Phi_s := \quad \forall n, x, y\ (\forall u\ \Phi[x, u]) \rightarrow \Phi[D(n, x), y]$$

$$C_s := \begin{cases} (1) & G(D(n, x)) \equiv F(n, G(x)) \\ (2) & H(D(n, x), y) \equiv D(n, H(x, y)) \\ (3) & F(F(x, y), z) \equiv F(x, F(y, z)) \end{cases}$$

Figure 1. Proof shell PS_s for the proof of φ_s (Simple Analysis)

If a new statement ψ shall be proved, a suitable induction axiom is selected by well-known automated methods, cf. (Walther 1994), from which a set of induction formulas I_ψ is computed for ψ. Then for proving an induction formula $\psi_i \in I_\psi$ by reuse, it is tested whether some proof shell PS *applies for* ψ_i, i.e. whether ψ_i is a (second-order) instance of the schematic induction formula of PS. If the test succeeds, the obtained (second-order) matcher is applied to the schematic catch of PS, and if all formulas of the instantiated schematic catch can be proved (which may necessitate further proof reuses), ψ_i is verified by reuse since the truth of an instantiated schematic catch implies the truth of its instantiated schematic induction formula.

For instance, assume that the new conjecture $\forall x, y\ \psi[x, y]$ shall be proved, where $\psi[x, y] :=$

$$\mathsf{times}(\mathsf{prod}(x), \mathsf{prod}(y)) \equiv \mathsf{prod}(\mathsf{app}(x, y))$$

and times and prod are defined by the axioms

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

The induction formulas computed for ψ are

$$\begin{array}{lll} \psi_b & := & \forall y\ \psi[\mathsf{empty}, y] \\ \psi_s & := & \forall n, x, y\ (\forall u\ \psi[x, u]) \rightarrow \psi[\mathsf{add}(n, x), y]. \end{array}$$

Obviously ψ is an instance of Φ and ψ_s is an instance of Φ_s w.r.t. the matcher $\pi := \{F/\mathsf{times}, G/\mathsf{prod}, H/\mathsf{app}, D/\mathsf{add}\}$. Hence (only considering the step case) we may reuse the given proof by instantiating the schematic catch C_s and subsequent verification of the resulting proof obligations $\pi(C_s) =$

$$\begin{array}{lll} (4) & \mathsf{prod}(\mathsf{add}(n, x)) & \equiv \mathsf{times}(n, \mathsf{prod}(x)) \\ (5) & \mathsf{app}(\mathsf{add}(n, x), y) & \equiv \mathsf{add}(n, \mathsf{app}(x, y)) \\ (6) & \mathsf{times}(\mathsf{times}(x, y), z) & \equiv \mathsf{times}(x, \mathsf{times}(y, z)) \end{array}$$

Features (4) and (5) are axioms, viz. (prod-2) and (app-2), and therefore are obviously true. So it only remains to prove the associativity of times (6) and, if successful, ψ_s is proved. Compared to a direct proof of ψ_s we have saved the user interactions necessary to apply the right axioms in the right place (where the associativity of times must be verified in either case). Additionally, conjecture (6) has been *speculated as a lemma* which is required for proving conjecture ψ.

The Phases of the Reuse Process

Our approach for reusing proofs is organized into the following steps, cf. Figure 2:

[5] Not to be confused with *generalization* of a formula φ as a preprocessing for proving φ by induction.

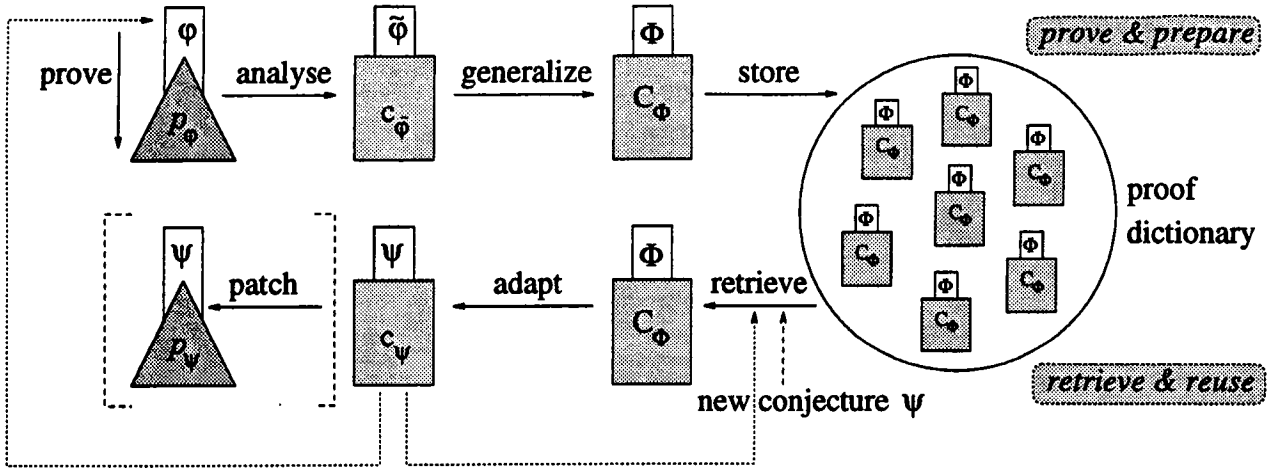

Figure 2. The Reuse Process

Prove: [cf. Sections 1, 2] If required, a *direct* proof p for (an induction formula) φ from a set of axioms AX is given by the human advisor or an automated induction theorem prover. The set of axioms AX consists of *defining* axioms, previously proved *lemmata*, and logical axioms like $x \equiv x$, $\varphi \rightarrow \varphi$ etc.

Analyze: (Kolbe & Walther 1994) The *simple proof analysis* which was illustrated in Section 2 analyzes a proof p of φ, yielding a *proof catch* c. Formally, the catch c is a finite subset of the non-logical axioms of AX such that c logically implies φ. For increasing the applicability of proof shells and the reusability of proofs, we have developed the *refined proof analysis* which also distinguishes different *occurrences* of function symbols in the conjecture and in the catch of a proof. For instance the (step formula of) statement $\psi_2 \quad := \quad \mathsf{plus}(\mathsf{len}(x), \mathsf{len}(y)) \equiv \mathsf{len}(\mathsf{app}(x,y))$ cannot be proved by reusing the proof shell from Figure 1, because one formula of the instantiated catch does not hold, cf. (Kolbe & Walther 1994). However, the reuse succeeds if refined analysis is applied (see below).

Generalize: (Kolbe & Walther 1994) Both φ and c are *generalized* by replacing (different occurrences of) function *symbols* with (different) function *variables*. This yields a *schematic conjecture* Φ and a *schematic catch* C, where the latter is a set of schematic formulas which — if considered as a set of first-order hypotheses — logically implies the schematic conjecture Φ. Such a pair $PS := \langle \Phi, C \rangle$ is called a *proof shell* and serves as the data structure for reusing the proof p. E.g. after the refined analysis of the proof of φ_s from Section 2, generalization yields $\Phi'[x,y] \quad := \quad F^1(G^1(x), G^2(y)) \equiv G^3(H^1(x,y))$ and the proof shell of Figure 3. Here e.g. the function variables F^1, F^2, F^3 correspond to different occurrences of the function symbol plus, i.e. the schematic equation (10) stems from generalizing (lem-1).

$$\Phi'_s := (\forall u\ F^1(G^1(x), G^2(u)) \equiv G^3(H^1(x,u))) \rightarrow F^1(G^1(D^1(n,x)), G^2(y)) \equiv G^3(H^1(D^1(n,x), y))$$

$$C'_s := \begin{cases} (7) & G^1(D^1(n,x)) \equiv F^2(n, G^1(x)) \\ (8) & H^1(D^1(n,x), y) \equiv D^4(n, H^1(x,y)) \\ (9) & G^3(D^4(n,x)) \equiv F^3(n, G^3(x)) \\ (10) & F^1(F^2(x,y), z) \equiv F^3(x, F^1(y,z)) \end{cases}$$

Figure 3. Proof shell PS'_s for the proof of φ_s (Refined Analysis)[6]

Store: (Kolbe & Walther 1995c) Proofs shells $\langle \Phi, C_1 \rangle$, ... ,$\langle \Phi, C_n \rangle$ (sharing a common schematic goal formula Φ) are merged into a *proof volume* $PV \ := \ \langle \Phi, \{C_1, ..., C_n\} \rangle$ which then is stored in the *proof dictionary* PD, i.e. a library of "proof ideas" organized as a set of proof volumes.

Retrieve: (Kolbe & Walther 1995c) If a new conjecture ψ is to be proved, the proof dic-

[6]Note that corresponding function variables in the induction hypothesis resp. the induction conclusion have been identified during the analysis phase.

tionary is searched for a proof volume $PV := \langle \Phi, \{C_1, ..., C_n\} \rangle$ such that $\psi = \pi(\Phi)$ for some second-order matcher π. For instance, $\pi_2 := \{F^1/\mathsf{plus}, G^{1,2,3}/\mathsf{len}, H^1/\mathsf{app}, D^{1,2}/\mathsf{add}\}$ is obtained by matching Φ'_s from Figure 3 with ψ_2 above. If successful, the schematic conjecture Φ and in turn also the proof volume PV *applies for* ψ (*via the matcher* π). Then a catch C_i is selected by heuristic support from the proof volume PV and the *partially instantiated catch* $\pi(C_i)$ serves as a candidate for proving ψ by reuse. For our example, the partially instantiated catch is obtained as $\pi_2(C'_s) =$

$$
\begin{array}{lll}
(11) & \mathsf{len}(\mathsf{add}(n,x)) & \equiv F^2(n, \mathsf{len}(x)) \\
(12) & \mathsf{app}(\mathsf{add}(n,x),y) & \equiv D^4(n, \mathsf{app}(x,y)) \\
(13) & \mathsf{len}(D^4(n,x)) & \equiv F^3(n, \mathsf{len}(x)) \\
(14) & \mathsf{plus}(F^2(x,y),z) & \equiv F^3(x, \mathsf{plus}(y,z))
\end{array}
$$

Adapt: (Kolbe & Walther 1995a) Since a partially instantiated catch $\pi(C_i)$ may contain *free function variables*, i.e. function variables which occur in C_i but not in Φ, these function variables have to be instantiated by known functions. Free function variables as F^2, F^3, and D^4 in $\pi_2(C'_s)$ result from the refined analysis and provide an increased flexibility of the approach, because different instantiations correspond to different proofs. Hence a further second-order substitution ρ is required for replacing these function variables such that the resulting proof obligations, i.e. all formulas in the *totally instantiated catch* $\rho(\pi(C))$, are provable from AX. Such a second-order substitution ρ is called a *solution* (for the free function variables), and ψ is proved by reuse because semantical entailment is invariant w.r.t. (second-order) instantiation. Solutions ρ are computed by *second-order matching modulo symbolical evaluation*, cf. (Kolbe & Walther 1995d). For the example, the solution $\rho_2 := \{F^{2,3}/\mathsf{s}(w_2), D^4/\mathsf{add}\}$ is obtained which e.g. instantiates (11) to the axiom $\mathsf{len}(\mathsf{add}(n,x)) \equiv \mathsf{s}(\mathsf{len}(x))$.[7]

Patch: (Kolbe & Walther 1995a) Often one is not only interested in the *provability* of ψ, but also in a *proof* of ψ which can be presented to a human or can be processed subsequently. In this case it is not sufficient just to *instantiate* the schematic proof P of Φ (which was obtained by generalizing the proof p of φ) with the computed substitution $\tau := \rho \circ \pi$ because τ might destroy the structure of P. Therefore the instantiated proof $\tau(P)$ is *patched* (which always succeeds) by removing void resp. inserting additional inference steps for obtaining a proof p' of ψ, cf. (Kolbe & Walther 1995b).

[7] The instantiations of F^1 and F^2 are different here, viz. plus and s, and this is why reuse fails for the simple analysis, cf. Figure 1.

Apart from initial proofs provided by the human advisor in the "prove" step, none of these steps necessitates human support. Thus the proof shell from Figure 3 can be automatically reused for proving the step formulas φ'_i of the *apparently different* conjectures φ_i given in Table 1 below.

This table illustrates a typical session with the PLAGIATOR-system: At the beginning of the session the human advisor submits statement φ (in the first row) and a proof p of φ to the system. Then the statements $\varphi_0, \varphi_1, \ldots, \varphi_5$ are presented to the PLAGIATOR, which proves each statement (and also statements $\varphi_6, \ldots, \varphi_{11}$ obtained as speculated lemmata) only by reuse of p such that no user interactions are required. The third column shows the subgoals speculated by the system when proving a statement by reuse. E.g. statement φ_6 is speculated when verifying φ_2, which leads to speculating φ_7 which in turn entails speculation of conjecture φ_8, for which eventually φ_9 is speculated.

φ	$\sum x + \sum y \equiv \sum(x <> y)$	φ_9
Φ	$F^1(G^1(x), G^2(y)) \equiv G^3(H^1(x,y))$	
conjectures proved by reuse		
φ_0	$\lvert x \rvert + \lvert y \rvert \equiv \lvert x <> y \rvert$	–
φ_1	$2x + 2y \equiv 2(x+y)$	–
φ_2	$(\mathsf{z}^y)^x \equiv \mathsf{z}^{x \times y}$	φ_6
φ_3	$\prod x \times \prod y \equiv \prod(x <> y)$	φ_7
φ_4	$x + y \equiv y + x$	φ_{10}
φ_5	$\mathsf{rev}(y) <> \mathsf{rev}(x) \equiv \mathsf{rev}(x <> y)$	$\sigma(\varphi_{11})$
φ_6	$\mathsf{z}^x \times \mathsf{z}^y \equiv \mathsf{z}^{x+y}$	φ_7
φ_7	$x \times (\mathsf{y} \times \mathsf{z}) \equiv (x \times \mathsf{y}) \times \mathsf{z}$	φ_8
φ_8	$x \times \mathsf{z} + y \times \mathsf{z} \equiv (x+y) \times \mathsf{z}$	φ_9
φ_9	$x + (y + z) \equiv (x+y) + \mathsf{z}$	–
φ_{10}	$x + \mathsf{s}(y) \equiv \mathsf{s}(x+y)$	–
φ_{11}	$x <> (y <> z) \equiv (x <> y) <> \mathsf{z}$	–

Table 1. Conjectures proved by reuse

Benefits of Reuse

The success of a reuse system (like the one presented in the previous sections) depends on the properties of the underlying problem solver and the domain which it is operating in:

The most drastic gains of reuse are obtained if there is no basic problem solver at all: Imagine a domain which is too complex or where too many uncertainties are involved for developing a systematic from-scratch problem solver. All what can be done is to collect some pairs of problems and their solutions by non-systematic means such as experiments and observations, e.g. for identification, classification or prediction tasks. Here reuse is the only way of solving future problems, and obviously all potential benefits can be obtained in this case, cf. *case-based reasoning* techniques (Kolodner 1993, Watson 1995).

In domains where the problem solver is a human or a machine supported considerably by user interactions, the most important benefit of reuse is saving resources, i.e. replacing user interactions by automated reuse of decisions contributing to a solution, cf. e.g. (Reif & Stenzel 1993). This can be seen as *programming by demonstration*, because providing a reuse system with typical examples is suitable to invoke resp. improve its problem solving behavior. But also improving the performance resp. the solution quality applies here as some solutions for similar problems can be proposed to the human for supporting his/her problem solving decisions in complex domains.

If the problem solver is a machine, a minor improvement in problem solving time cannot justify the additional effort for building reuse systems, in particular as in some scenarios reuse can be *proven* to perform *worse* than problem solving from scratch, cf. (Kolbe & Walther 1996) and also (Nebel & Koehler 1993). Hence the saving of (the generally cheap) resources is only relevant if the system operates with limited resources (e.g. time-critical systems in real-time applications). However, improving an incomplete system's performance is an important potential benefit which seems not to be well recognized in the literature. The quality of solutions produced by a machine often depends mainly on the amount of time spent for improving found or searching for better solutions, hence the benefits are the same as with saving resources.

Consequences for Reuse Systems

An analysis of the benefits of reuse reveals that, contrary to the often adopted position, *speedup* is not the only resp. not the most important benefit of reusing past problem solutions (if it is obtained at all), cf. (Kolbe & Walther 1996). The benefits rather depend on the application domain and the kind of problem solver which is available. Therefore the design of a reuse system should be influenced by this analysis:

If the problem solver is a human or a machine supported considerably by user interactions, the system should be designed to achieve *high reusability*, because the savings of (even a few) human interactions will outweigh the costs of (even a large amount of) machine resources (wrt. time and memory).

If the problem solver is a machine working in a real-time environment, the reuse system has to be tailored to optimize retrieval and adaptation time, even if the degree of reusability is decreased.

Without severe time-constraints, incorporating reuse in automated problem solvers pays only in domains where *drastic* savings are achievable, or where an *incomplete* problem solver can be improved by providing additional solutions by reuse.

Conclusion

We have presented an approach for reusing proofs which is implemented with the PLAGIATOR-system (Brauburger 1994). Experiments with the system reveal that *automated* proof reuse is successful even for conjectures which differ considerably in both their structure and their domain. Therefore we claim that the complex task of automated reasoning can be learned from a human advisor. For achieving high reusability, sophisticated techniques with powerful heuristics have been developed to gain the high flexibility which is required for retrieval and adaptation of known proofs. Future work shall deepen the practical evaluation of our approach by analysis of additional experiments with more complex reasoning problems.

References

Bouhoula, A.; Kounalis, E.; and Rusinowitch, M. 1992. SPIKE: An Automatic Theorem Prover. In *Proceedings of the Conference on Logic Programming and Automated Reasoning,* St. Petersburg, Russia. Springer.

Boyer, R. S., and Moore, J. S. 1979. *A Computational Logic.* ACM Monograph Series. Academic Press.

Brauburger, J. 1994. PLAGIATOR: Design and Implementation of a Learning Theorem Prover. Diploma Thesis (in German), TH Darmstadt.

Bundy, A.; van Harmelen, F.; Horn, C.; and Smaill, A. 1990. The Oyster-Clam System. In *Proceedings of the 10th International Conference on Automated Deduction,* Kaiserslautern, Germany, 647–648.

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

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

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

Kapur, D., and Zhang, H. 1988. RRL: A Rewrite Rule Laborarory. In Lusk, E., and Overbeek, R., eds., *Proceedings of the 9th International Conference on Automated Deduction,* Argonne, number 310 in LNCS. Springer. 768–769.

Kolbe, T., and Walther, C. 1994. Reusing Proofs. In Cohn, A., ed., *Proceedings of the 11th European Conference on Artificial Intelligence, Amsterdam.* John Wiley & Sons, Ltd. 80–84.

Kolbe, T., and Walther, C. 1995a. Adaptation of Proofs for Reuse. In Aha, D. W., and Ram, A., eds., *Working Notes of the 1995 AAAI Fall Symposium on Adaptation of Knowledge for Reuse,* 61–67.

Kolbe, T., and Walther, C. 1995b. Patching Proofs for Reuse. In Lavrac, N., and Wrobel, S., eds., *Proc. European Conf. on Machine Learning (ECML-95),* 303 – 306.

Kolbe, T., and Walther, C. 1995c. Proof Management and Retrieval. In *Proceedings of the IJCAI'95 Workshop on* Formal Approaches to the Reuse of Plans, Proofs, and Programs.

Kolbe, T., and Walther, C. 1995d. Second-Order Matching modulo Evaluation — A Technique for Reusing Proofs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence,* Montreal, Canada.

Kolbe, T., and Walther, C. 1996. On the Benefits of Reusing Past Problem Solutions. Technical Report, TH Darmstadt.

Kolodner, J. L. 1993. *Case-Based Reasoning.* San Mateo, CA: Morgan Kaufmann.

Nebel, B., and Koehler, J. 1993. Plan Modification versus Plan Generation: A Complexity-Theoretic Perspective. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence,* Chambery, France, 1436–1441.

Nilsson, N. J. 1971. *Problem Solving Methods in Artificial Intelligence.* McGraw Hill, New York.

Reif, W., and Stenzel, K. 1993. Reuse of Proofs in Software Verification. In Shyamasundar, R., ed., *Foundation of Software Technology and Theoretical Computer Science,* Bombay, India.

Villafiorita, A., and Giunchiglia, F. 1996. Inductive Theorem Proving via Abstraction. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics.* To appear.

Walther, C. 1994. Mathematical Induction. In Gabbay, D. M.; Hogger, C. J.; and Robinson, J. A., eds., *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 2. Oxford University Press. 127–227.

Watson, I. D. 1995. An Introduction to Case-Based Reasoning. In Watson, I. D., ed., *Progress in Case-Based Reasoning.* Springer LNAI 1020. 3–16.