Second-Order Matching modulo Evaluation — A Technique for Reusing Proofs

Thomas Kolbe   Christoph Walther
FB Informatik, TH Darmstadt, Alexanderstr. 10, D-64283 Darmstadt, Germany,
e-mail: {kolbe|walther}@inferenzsysteme.informatik.th-darmstadt.de

Abstract[1]

We investigate the improvement of theorem provers by reusing previously computed proofs. A proof of a conjecture is generalized by replacing function symbols with *function variables*. This yields a *schematic* proof of a *schematic* conjecture which is instantiated subsequently for obtaining proofs of new, similar conjectures. Our reuse method requires *solving* so-called *free* function variables, i.e. variables which cannot be instantiated by matching the schematic conjecture with a new conjecture. We develop an algorithm for solving free function variables by combining the techniques of *symbolic evaluation* and *second-order matching*. Heuristics for controlling the algorithm are presented, and several examples demonstrate their usefulness. We also show how our reuse proposal supports the discovery of useful lemmata.

1 Introduction

We investigate the improvement of theorem provers by reusing previously computed proofs, cf. [Kolbe and Walther, 1994; 1995]. Our work has similarities with the machine learning methodologies of *explanation-based learning* [Ellman, 1989], *analogical reasoning* [Hall, 1989], and *abstraction* [Giunchiglia and Walsh, 1992]. An abstract view of our approach for reusing proofs can be sketched in the following way:

Assume that an automated theorem prover shall be supplemented by a learning component. If the prover is asked to prove a new conjecture ψ which is *similar* to a previously proven conjecture φ, it first tries to associate each function symbol occurring in φ with a (syntactically) corresponding function symbol from ψ. These associations are then propagated into the *proof* of φ for obtaining a proof of ψ. But generally, there are function symbols in the *proof* of φ which do not occur in the *conjecture* φ and thus still have to be associated. If these function symbols can be associated such that the axioms used in the proof of φ are mapped to provable formulas, the proof of φ is successfully *reused* for proving ψ.

[1]This work was supported under grants no. Wa652/4-1,2,3 by the DFG as part of the focus program "Deduktion".

Here we are concerned with providing the associations for function symbols automatically such that verifiable proof obligations for ψ are obtained.

We propose an indirect way of association by first *generalizing* the function symbols from (the proof of) φ to *function variables*, i.e. we use a second-order language. Then we *instantiate* the function variables occurring in the generalization of φ — the so-called *bound* function variables — with function symbols by second-order matching with ψ. For providing the remaining associations a further second-order substitution has to be found which replaces the so-called *free* function variables, i.e. function variables which occur in the generalization of the proof of φ, but not in the generalization of φ. We call such a second-order substitution a *solution* (for the free function variables), if all formulas stemming from axioms in the proof of φ are true.

As the solvability of free function variables is defined w.r.t. a set of axioms, it is undecidable in general whether a solution exists, and in the positive case the solution is not unique. Here we develop an efficient procedure for computing solutions by *second-order matching "modulo evaluation"*. Our algorithm incorporates the underlying axioms by heuristically combining the second-order matching algorithm of [Huet and Lang, 1978] with the technique of *symbolic evaluation*, cf. e.g. [Walther, 1994].

2 A Method for Reusing Proofs

We briefly illustrate our technique for reusing proofs, cf. [Kolbe and Walther, 1994] for a more detailed account: If the prover has computed a proof p for a conjecture φ, the proof is *analyzed* yielding a so-called *proof catch* c. The catch is a set of axioms providing the features of p which are relevant for reusing the proof in subsequent verification tasks. Then φ and c are *generalized* by replacing (different occurrences of) function symbols with (different) function variables, yielding a *schematic conjecture* Φ and a *schematic catch* C. The latter is a set of schematic formulas which logically implies the schematic conjecture Φ. Thus $C \models \Phi$ is guaranteed by our analysis and generalization method where $\models$ denotes (first-order) semantical entailment. Such a pair $PS := \langle \Phi, C \rangle$ is called a *proof shell* and serves as a base for reusing the proof p if the system has to prove a new conjecture ψ which is *similar* to φ:

If ψ is obtained from Φ by matching, i.e. $\psi = \pi(\Phi)$ for a second-order substitution π (which means that ψ is *similar* to φ), then the proof shell PS *applies for* ψ *via the matcher* π. Here π replaces the function variables in Φ – the so-called *bound function variables* of PS – by function symbols (resp. by functional terms, cf. Section 5). Now we apply π to the schematic catch C obtaining the *partially instantiated* schematic catch $\pi(C)$ which still may contain function variables, viz. the *free function variables* of PS occurring in C, but not in Φ. Now a further second-order substitution ρ has to be found which replaces the free function variables of PS such that all formulas in the *instantiated schematic catch* $\rho(\pi(C))$ are provable. If successful then also ψ is proved because $\rho(\pi(C)) \models \rho(\pi(\Phi))$ and $\rho(\pi(\Phi)) = \rho(\psi) = \psi$, which means that the proof p of φ could be reused for proving ψ. Thus verifying the *resulting proof obligations* $\rho(\pi(C))$ is sufficient for guaranteeing the provability of ψ. This approach is sound because semantical entailment is invariant w.r.t. (second-order) instantiation, i.e. $C \models \Phi$ implies $\tau(C) \models \tau(\Phi)$ for each second-order substitution τ, cf. [Kolbe and Walther, 1995].

3 Matching modulo Evaluation

We investigate our proposal within the field of automated *mathematical induction*, cf. e.g. [Walther, 1994]. Here proofs often are similar, and consequently automated induction provides the regularity required for successful reuses. For proving a given conjecture, an induction theorem prover computes a set of *induction formulas* whose provability imply the truth of the conjecture. Then the system tries to prove each induction formula from a set of axioms by first-order means, and the proofs obtained thereby are candidates for subsequent reuses.

We define functions like the length function len for linear lists by axioms like

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

Given the conjecture $\psi := \mathsf{plus}(\mathsf{len}(x), \mathsf{len}(y)) \equiv \mathsf{len}(\mathsf{app}(x,y))$, the induction theorem prover computes a base formula ψ_b and a step formula ψ_s as (plus denotes addition of natural numbers, app denotes list concatenation, sum denotes summation over a linear list):

$$\psi_b := \mathsf{plus}(\mathsf{len}(\mathsf{empty}), \mathsf{len}(y)) \equiv \mathsf{len}(\mathsf{app}(\mathsf{empty}, y))$$

$$\psi_s := (\forall u\ \mathsf{plus}(\mathsf{len}(x), \mathsf{len}(u)) \equiv \mathsf{len}(\mathsf{app}(x,u))) \rightarrow \\ \mathsf{plus}(\mathsf{len}(\mathsf{add}(n,x)), \mathsf{len}(y)) \equiv \mathsf{len}(\mathsf{app}(\mathsf{add}(n,x),y))$$

Assume that the following proof shell PS is provided which was obtained by analyzing and generalizing the proof of the step formula φ_s for the conjecture φ, cf. [Kolbe and Walther, 1994], where

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

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

$$C_s := \left\{ \begin{array}{lrcl} (1) & G^1(D^1(n,x)) & \equiv & F^2(n, G^1(x)) \\ (2) & H^1(D^1(n,x),y) & \equiv & D^4(n, H^1(x,y)) \\ (3) & G^3(D^4(n,x)) & \equiv & F^3(n, G^3(x)) \\ (4) & F^1(F^2(x,y),z) & \equiv & F^3(x, F^1(y,z)) \end{array} \right\}$$

Figure 1. The proof shell PS for the proof of φ_s

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 (4) stems from generalizing the lemma $\mathsf{plus}(\mathsf{plus}(x,y),z) \equiv \mathsf{plus}(x, \mathsf{plus}(y,z))$ which was required for proving φ_s. As PS applies for ψ_s via the matcher $\pi_s := \{F^1/\mathsf{plus}, G^{1,2,3}/\mathsf{len}, H^1/\mathsf{app}, D^1/\mathsf{add}\}$, the partially instantiated catch $C'_s := \pi_s(C_s) =$

$$\left\{ \begin{array}{lrcl} (5) & \mathsf{len}(\mathsf{add}(n,x)) & \equiv & F^2(n, \mathsf{len}(x)) \\ (6) & \mathsf{app}(\mathsf{add}(n,x),y) & \equiv & D^4(n, \mathsf{app}(x,y)) \\ (7) & \mathsf{len}(D^4(n,x)) & \equiv & F^3(n, \mathsf{len}(x)) \\ (8) & \mathsf{plus}(F^2(x,y),z) & \equiv & F^3(x, \mathsf{plus}(y,z)) \end{array} \right\}$$

is computed for the step case. The free function variables F^2, F^3 and D^4 stemming from (occurrences of) the function symbols plus and add in the catch of the original proof are not replaced by concrete function symbols: Since these function variables do not occur in the schematic conjecture Φ_s, they cannot be in the domain of the matcher π_s. We therefore have to *compute* a second-order substitution ρ_s for these free function variables which *solves* C'_s, i.e. we are looking for some ρ_s such that the formulas $\rho_s(C'_s)$ are provable. The attempt of solving F^2 in (5) by (syntactical) second-order *matching* fails, but we may *evaluate* the left hand side (lhs) of (5) by the defining axiom (len-2) yielding $\mathsf{s}(\mathsf{len}(x)) \equiv F^2(n, \mathsf{len}(x))$. Now the lhs matches the rhs via the unique matcher $\rho_1 := \{F^2/\mathsf{s}(w_2)\}$ which means that (5) is solved because $\rho_1((5)) = (\mathsf{len\text{-}2})$.[2] We apply ρ_1 to the remaining schematic formulas and obtain $\rho_1(C'_s \setminus \{(5)\})$ as

$$\left\{ \begin{array}{lrcl} (6) & \mathsf{app}(\mathsf{add}(n,x),y) & \equiv & D^4(n, \mathsf{app}(x,y)) \\ (7) & \mathsf{len}(D^4(n,x)) & \equiv & F^3(n, \mathsf{len}(x)) \\ (9) & \mathsf{plus}(\mathsf{s}(y),z) & \equiv & F^3(x, \mathsf{plus}(y,z)) \end{array} \right\}.$$

We continue with the next schematic equation whose (say) lhs does not contain function variables, i.e. with (6) or with (9). Here we choose (6) and evaluating its lhs using a defining axiom for app yields $\mathsf{add}(n, \mathsf{app}(x,y))$ which matches the rhs via the unique matcher $\rho_2 := \{D^4/\mathsf{add}\}$. Now the lhs of $\rho_2((7))$ is purely first-order and can be evaluated, i.e. $\rho_3 := \{F^3/\mathsf{s}(w_2)\}$ is obtained by matching. Finally all free function variables are instantiated and the remaining proof obligation $\rho_3((9))$ can be passed to the prover. As $\rho_3((9))$ is a defining axiom for plus, the prover is successful, i.e. the proof reuse is completed. Thus the alternation of *evaluation* and *second-order matching* allows us to *compute* the solution $\rho_s = \{F^2/\mathsf{s}(w_2), F^3/\mathsf{s}(w_2), D^4/\mathsf{add}\}$.[3]

4 Symbolic Evaluation

Let $\Sigma = \bigcup_{i \in \mathbb{N}} \Sigma_i$ be a signature where Σ_i contains (first-order) *function symbols* of arity i. We let $\mathcal{T}(\Sigma, \mathcal{V})$ denote the set of all *terms* built with function symbols from Σ and the set $\mathcal{V}$ of variables. The set of *equations* $t_1 \equiv t_2$ with $t_1, t_2 \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted by $\mathcal{E}(\Sigma, \mathcal{V})$, and *formulas*

[2] Here w_2 denotes the second argument of the binary function variable F^2, i.e. ρ_1 replaces F^2 basically with the function symbol s, but the first argument w_1 of F^2 is ignored.

[3] Note that the proposed method is more flexible than simply matching the schematic equations from C'_s with the defining axioms.

are defined as usual.[4] We denote the set of variables occurring in a (set of) term(s) resp. formula(s) ϕ by $\mathcal{V}(\phi)$.

We assume that each function symbol from Σ is either a *constructor* of some data structure or a *defined function symbol* specified by some axioms, cf. [Walther, 1994]. For instance the declaration

structure 0 s(number) : number

defines a data structure for natural numbers with the constructors 0 and s. The axioms (len-1,2) define the function symbol len, cf. Section 3. A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ may be *evaluated* by applying the defining axioms for the function symbols contained in t. Thus e.g. $\mathsf{len}(\mathsf{add}(\mathsf{len}(\mathsf{empty}), x))$ is evaluated via $\mathsf{len}(\mathsf{add}(0, x))$ to $\mathsf{s}(\mathsf{len}(x))$ by applying the defining axioms for len. This is called *symbolic* evaluation because the term may contain variables (which are not evaluated). Note that only defining axioms are considered for the evaluation, i.e. the term $\mathsf{s}(\mathsf{plus}(\mathsf{plus}(x, y), z))$ remains non-evaluable even if e.g. the associativity of plus is given as a lemma.

We assume that the symbolic evaluator is implemented by a *terminating* operation *eval* [Walther, 1994] which computes the normal form of a term such that

$$(10) \qquad eval(l) \in \mathcal{T}(\Sigma, \mathcal{V}), \quad AX \models l \equiv eval(l)$$

is satisfied for all $l \in \mathcal{T}(\Sigma, \mathcal{V})$ and a set AX of axioms and lemmata.

5 Second-Order Matching

Let $\Omega = \bigcup_{i \in \mathbb{N}} \Omega_i$ be a signature where Ω_i contains *function variables* of arity i. Then $\mathcal{T}(\Sigma \cup \Omega, \mathcal{V})$ denotes the set of all *schematic terms* and $\mathcal{E}(\Sigma \cup \Omega, \mathcal{V})$ denotes the set of all *schematic equations* built with function symbols from Σ, function variables from Ω, and variables from $\mathcal{V}$. *Schematic formulas* are built as usual,[5] and $\Omega(\phi)$ denotes the set of all function variables occurring in a (set of) schematic term(s) resp. formula(s) ϕ. Function variables are treated like function symbols except for the application of second-order substitutions.

For defining second-order substitutions we represent functions by terms with special *argument variables* from the set $\mathcal{W} := \{w_1, w_2, ...\}$ where $\mathcal{W} \cap \mathcal{V} = \emptyset$. Thus $\mathcal{T}(\Sigma, \mathcal{W})$ is the set of *functional terms* and we define a family $(\mathcal{W}_i)_{i \in \mathbb{N}}$ of subsets of $\mathcal{W}$ by $\mathcal{W}_0 := \emptyset$ and $\mathcal{W}_i := \{w_1, ..., w_i\}$. A *(restricted) second-order substitution* $\pi := \{..., F/s, ...\}$ is a finite set of pairs such that the *domain* $\mathrm{dom}(\pi) := \{..., F, ...\} \subseteq \Omega$ contains (distinct) function variables and $F \in \Omega_i$ entails $s \in \mathcal{T}(\Sigma, \mathcal{W}_i)$, i.e. a function variable F of arity i is replaced by a functional term s with at most i argument variables. The *application* of a restricted second-order substitution π to a schematic term is inductively defined by

$$\begin{aligned}
\pi(x) &:= x \\
\pi(X(P_1, ..., P_n)) &:= X(\pi(P_1), ..., \pi(P_n)) \\
\pi(F(P_1, ..., P_n)) &:= s[w_1/\pi(P_1), ..., w_n/\pi(P_n)]
\end{aligned}$$

[4]Our logic is *many-sorted* in general, but for the sake of readability we usually omit all sort information.

[5]Note that only first-order variables may be quantified.

where $x \in \mathcal{V}$, $X \in \Sigma_n \cup (\Omega_n \setminus \mathrm{dom}(\pi))$, $F \in \Omega_n \cap \mathrm{dom}(\pi)$, $\pi(F) = s$ and $P_1, ..., P_n \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V})$. E.g. the application of $\pi := \{F/\mathsf{len}(w_2), G/\mathsf{plus}(w_1, \mathsf{s}(w_1))\}$ to the schematic term $P := G(F(\mathsf{app}(x, y), z), v)$ yields $\pi(P) = \mathsf{plus}(\mathsf{len}(z), \mathsf{s}(\mathsf{len}(z)))$. For $f \in \Sigma_n$ and $F \in \Omega_n$ we use $\pi(F) = f$ as an abbreviation of $\pi(F) = f(w_1, ..., w_n)$. Restricted second-order substitutions are applied to sets of terms or (sets of) formulas as usual. Note that $\mathcal{V}(\pi(t)) \subseteq \mathcal{V}(t)$ and $\Omega(\pi(t)) \subseteq \Omega(t)$ for $t \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V})$, i.e. no first-order variables or function variables are introduced by restricted second-order substitutions. Furthermore $\mathrm{dom}(\pi) \subseteq \Omega$, i.e. no (first-order) variables are replaced. These restrictions (compared to general second-order substitutions, cf. [Goldfarb, 1981]) are necessary to ensure the soundness of our method for reusing proofs, cf. Section 3. As a consequence the subsumption quasi-ordering $\preceq$ on general second-order substitutions from [Huet and Lang, 1978] corresponds to the subset relation $\subseteq$ in our context: $\preceq$ is defined by $\pi \preceq \pi'$ iff $\exists \rho\ \pi' = \rho \circ \pi$ where $\circ$ denotes the composition of general second-order substitutions. But as $F \in \mathrm{dom}(\pi)$ implies $\rho(\pi(F)) = \pi(F)$ for restricted second-order substitutions ρ, π we have $\preceq\, = \,\subseteq$ here. Subsequently we simply say "second-order substitution" instead of "restricted second-order substitution".

A target term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ *matches* a pattern $P \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V})$ iff there is a second-order substitution π with $t = \pi(P)$. E.g. the target $t \equiv \mathsf{s}(\mathsf{sum}(x))$ matches the pattern $P \equiv F(x, \mathsf{sum}(x))$ because the required matcher exists, viz. $\pi = \{F/\mathsf{s}(w_2)\}$ or $\pi = \{F/\mathsf{s}(\mathsf{sum}(w_1))\}$. Second-order matching is decidable and we can adapt the algorithm from [Huet and Lang, 1978] for computing a finite set $\Pi := match(t, P)$ of matchers satisfying the following conditions:

consistence	$\pi \in \Pi \Rightarrow t = \pi(P)$	(11)
completeness	$t = \pi'(P) \Rightarrow \exists\, \pi \in \Pi.\ \pi \subseteq \pi'$	(12)
minimality	$\pi, \pi' \in \Pi,\ \pi \neq \pi' \Rightarrow \pi' \not\subseteq \pi$	(13)

6 Solving a Proof Catch

We present the algorithm *solve_catch* which implements the second-order matching "modulo evaluation" illustrated in Section 3. We assume a fixed set of (implicitly universally quantified) axioms $AX \subseteq \mathcal{E}(\Sigma, \mathcal{V})$ in the remainder of this section. The algorithm *solve_catch* has two input arguments, viz. a partially instantiated schematic catch $C' \subseteq \mathcal{E}(\Sigma \cup \Omega, \mathcal{V})$ and an auxiliary set of schematic equations $X \subseteq \mathcal{E}(\Sigma \cup \Omega, \mathcal{V})$ which initially is empty. The call *solve_catch*$(C', \emptyset)$ *fails* if no matcher solving C' has been found. Otherwise it yields a pair $\langle \rho, E \rangle$ where ρ is a second-order substitution and $E \subseteq \mathcal{E}(\Sigma, \mathcal{V})$ is a set of (first-order) equations such that

$$E \subseteq \rho(C') \subseteq \mathcal{E}(\Sigma, \mathcal{V}) \quad \text{and} \quad AX \cup E \models \rho(C'). \qquad (14)$$

This means that the schematic catch C' is solved by the matcher ρ, if the call *solve_catch*$(C', \emptyset)$ yields a pair $\langle \rho, E \rangle$ such that the remaining proof obligations in E can be verified. We start with the algorithm for *solve_catch* and then discuss the used auxiliary functions:[6]

[6]The special value Λ indicates that it has been tried to select a member from an empty set, see below.

```
**function** *solve_catch* $(C', X)$
$l \equiv R := choose\_eq(C')$
**if** $l \not\equiv \Lambda$ **then**
  $\Pi := match(eval(l), R)$
  **if** $(\pi := choose\_matcher(\Pi)) = \Lambda$ **then**
    **return** $solve\_catch(C' \setminus \{l \equiv R\}, X \cup \{l \equiv R\})$
  **else**
    $\langle \rho, E \rangle := solve\_catch(\pi(C' \setminus \{l \equiv R\}), \pi(X))$
    **return** $\langle \pi \cup \rho, E \rangle$ **fi**
**else**
  **if** $\Omega(C' \cup X) = \emptyset$ **then return** $\langle \{\}, C' \cup X \rangle$
  **else fail fi fi**
```

The function *choose_eq* selects some equation $l \equiv R$ from C', whose (say) lhs l does *not* contain function variables, but whose rhs R *does* contain function variables.[7] We demand that for all $A \subseteq \mathcal{E}(\Sigma \cup \Omega, \mathcal{V})$

(15) $M_A = \emptyset$ iff $choose\_eq(A) = \Lambda \equiv \Lambda$

(16) $M_A \neq \emptyset$ implies $choose\_eq(A) \in M_A$

where $M_A := \{l \equiv R \in A \mid \Omega(l) = \emptyset, \Omega(R) \neq \emptyset\}$. The heuristic used by *choose_eq* for selecting the equation $l \equiv R \in M_{C'}$ is discussed in the next section. The term l is *evaluated* by applying one or more defining axioms from AX until a term $l' := eval(l)$ is obtained, cf. (10). Then the second-order matching algorithm *match* is called for l' and R, and all computed matchers are collected in a set Π. Of course we require that this set of matchers $\Pi := match(l', R)$ is *consistent*, cf. (11), but we do not demand *completeness* as we later select only one matcher from it anyway. Instead of (13), however, we now request a stronger form of minimality which takes the lack of completeness into account:

(17) $\pi \in \Pi \wedge l = \pi'(R) \wedge \pi \neq \pi'$ implies $\pi' \not\subseteq \pi$

From the computed set of matchers Π, *choose_matcher* selects one matcher (if Π is non-empty) due to some heuristic, cf. the next section.

If $\Pi = \emptyset$, no solution for the free function variables in R was found by matching modulo evaluation. Then $l \equiv R$ is inserted into the so-called *remainder set* X (which initially is empty) and we continue by solving the remaining part of the catch $C' \setminus \{l \equiv R\}$. We will not try to solve the equations collected in X by matching again (as this must fail), but we hope that the function variables occurring in X are instantiated while solving the remaining catch such that a set of first-order proof obligations is obtained from X. Then the desired matcher ρ and the set of first-order proof obligations E is returned, cf. (14). Otherwise the recursive call (and thus the whole procedure) fails.

If some $\pi \in \Pi$ is selected, we apply π to the remaining part of the catch $C' \setminus \{l \equiv R\}$ as well as to the remainder set X and recursively call *solve_catch*. If the recursive call does not fail, we obtain a matcher ρ and a set of first-order proof obligations E. Then $\pi \cup \rho$ is the resulting matcher for solving the catch C' (note that $\mathrm{dom}(\pi) \cap \mathrm{dom}(\rho) = \emptyset$ as π is applied before the recursive call) which is returned together with E.

[7]Without loss of generality we assume that the equations are oriented such that there is no equation $L \equiv r$ with $\Omega(L) \neq \emptyset$ and $\Omega(r) = \emptyset$.

If a recursive call of *solve_catch* is reached where *choose_eq* yields Λ, i.e. all equations in $C' \setminus \mathcal{E}(\Sigma, \mathcal{V})$ (if any) contain function variables on *both* sides, cf. (15), then the alternating process of matching modulo evaluation and instantiating the catch terminates. Now if $\Omega(C' \cup X) = \emptyset$ then the catch is solved by the identity substitution $\{\}$, and $C' \cup X$ is returned as the set of first-order proof obligations. In this case also the initial call of *solve_catch* is successful by collecting the matchers which have been computed in the respective recursive calls. Otherwise there is an equation in C' with function variables on both sides (which has not yet been processed) or there is an equation in X with function variables on one side (where the matching failed). We regard the catch as unsolvable then and *solve_catch* fails, although one can think of several alternative heuristically motivated attempts (cf. Section 7) of solving the remaining function variables. We summarize these considerations with the following theorem:

Theorem 6.1 (soundness of *solve_catch*) *Let $AX \subseteq \mathcal{E}(\Sigma, \mathcal{V})$ be a set of axioms. If the conditions (10), (11), (15) and (16) are satisfied, then for all schematic catches C' and $X \subseteq \mathcal{E}(\Sigma \cup \Omega, \mathcal{V})$ the call $solve\_catch(C', X)$ either fails or it terminates with result $\langle \rho, E \rangle$ such that*

$$E \subseteq \rho(C' \cup X) \subseteq \mathcal{E}(\Sigma, \mathcal{V}) \text{ and } AX \cup E \models \rho(C' \cup X).$$

The function *solve_catch* is designed for an efficient computation of a solution for a partially instantiated schematic catch $\pi(C)$. If there is no such solution, i.e. the reuse attempt fails, this might lead to backtracking in the calling procedure, e.g. for choosing a different π or C. Hence we are interested in an *early failure recognition* within *solve_catch*. This can be achieved if we try to *disprove* those equations $e \in C' \cup X$ which do not contain function variables by searching for a counterexample, cf. [Protzen, 1992], yielding one of the results *disproved*, *proved* or *unknown*. If e is disproved, then *solve_catch* fails as the matcher computed so far yields unprovable proof obligations, i.e. we can save any effort for solving the remaining function variables. If e is proved, it can be removed (this might happen for some "simple" equations e), and otherwise e has to be verified after all function variables are solved.

7 Heuristics

The function *solve_catch* has two indeterministic *branching points*, viz. the calls of the functions *choose_eq* and *choose_matcher*, and some heuristic guidance is required for making a good choice. We have developed heuristics with empirical data obtained from several experiments. Although the branching points define an entry for *backtracking* whenever a failure in some recursive call occurs, we avoid backtracking as we believe that only a non-expensive solution procedure leads to useful proof reuses. Hence the success of our proposal depends on the quality of the developed heuristics.

The heuristic for *choose_eq* selects a schematic equation $l \equiv R \in M_{C'}$ which has probably the least number of matchers π solving $eval(l) = \pi(R)$, i.e. the most "difficult" of the equations (without function variables on the lhs) is chosen. This choice has two advantages: (i) the possibility for selecting a "wrong" matcher by *choose_matcher* is decreased because a point of the search space with a small branching rate is reached where it is more likely to find a good path, and (ii) the solvability of the "difficult" equation might be lost if another, "easier" equation is chosen where it is harder to distinguish a successful matcher from an unsuccessful one.

As a measure for the difficulty of (solving) an equation we consider the number of function symbols and function variables occurring in it: A high number of (occurrences of) function symbols limits the number of possible matchers in the same way as a low number of (different) function variables. Hence we define two mappings $\#_\Sigma : \mathcal{T}(\Sigma \cup \Omega, \mathcal{V}) \to \mathbb{N}$ and $\#_\Omega : \mathcal{T}(\Sigma \cup \Omega, \mathcal{V}) \to \mathbb{N}$ where $\#_\Sigma(t)$ counts the number of occurrences of function symbols and $\#_\Omega(t)$ counts the number of different function variables in a schematic term $t \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V})$.

When rating the difficulty of an equation $l \equiv R$ we mainly have to consider the rhs R, as the lhs l is already first-order and will be evaluated yet before the matching such that it does not influence the number of solutions strongly. For the moment, we use the following heuristic for *choose_eq*(C'): Among the candidate equations $l \equiv R \in M_{C'}$, the one with the minimal value

$$h_1(l \equiv R) \ := \ 2\#_\Omega(R) - \#_\Sigma(R)$$

is selected. The factor 2 serves for weighting the number of function variables higher than the number of function symbols as the former has a stronger influence on the success of the matching. For instance, given $l := \mathsf{len}(\mathsf{add}(n, x))$, $R_1 := F(\mathsf{len}(G(x)))$ and $R_2 := F(G(x))$, we prefer the equation $l \equiv R_1$ to $l \equiv R_2$ because $h_1(l \equiv R_1) = 3 < 4 = h_1(l \equiv R_2)$. A more sophisticated rating might also consider the structure of l or R, the first-order variables in $l \equiv R$, or even the remaining equations, but we found this not necessary so far.

The heuristic for *choose_matcher* attempts to select the "simplest" matcher $\pi \in match(eval(l), R)$ solving the chosen equation $l \equiv R$. We consider π as "simple" if it is "close" to replacing function variables by function symbols, e.g. $\pi(F) = f(w_1, \ldots, w_n)$, while e.g. $\pi := \{F/\mathsf{plus}(w_2, \mathsf{times}(w_3, w_2)), \ldots\}$ would be considered as a more complex matcher. The underlying motivation is to select a matcher which preserves the given structure of the proof being reused, because it is more likely to find a valid instance of the schematic proof in this case.

As a measure for simplicity we consider two features of a matcher: (i) a small domain and (ii) a small number of introduced function symbols. Concerning (i), we demand that the function *match* satisfies requirement (17), i.e. no function variables are replaced unnecessarily (though this is not demanded in Theorem 6.1). The introduced function symbols (ii) are those occurring in the functional terms of π counted by

$$\#_\Sigma(\pi) := \sum_{F \in \mathrm{dom}(\pi)} \#_\Sigma(\pi(F))$$

where we prefer matchers π such that $\#_\Sigma(\pi)$ is minimal. For instance, given $l' = \mathsf{sum}(x)$ and $R = F(x, \mathsf{sum}(x))$, we prefer the matcher $\pi_1 := \{F/w_2\}$ to the matcher $\pi_2 := \{F/\mathsf{sum}(w_1)\}$ because $\#_\Sigma(\pi_1) = 0 < 1 = \#_\Sigma(\pi_2)$. So among the candidate matchers $\pi \in match(eval(l), R)$, *choose_matcher* selects the one with the minimal value $\#_\Sigma(\pi)$. For obtaining an efficient implementation we incorporate the selection of the matcher into the matching algorithm such that only matchers π with minimal measure $\#_\Sigma(\pi)$ are computed.

Finally we sketch another heuristical extension for *solve_catch*, viz. how to proceed if *choose_eq*$(C') = \Lambda$ but $\Omega(C' \cup X) \neq \emptyset$. In this case there is an equation in C' with function variables on both sides (which has not yet been processed) or there is an equation in X with function variables on one side (where the matching failed). Now instead of failing we might try to solve the remaining function variables in the following way: If e.g. F^i and F^j are free function variables from C' stemming from generalizing different occurrences of the *same* function symbol f in the original proof, then we instantiate F^i and F^j with the same functional term. Hence we stipulate $\rho(F^j) := \rho(F^i)$ if $F^i \in \mathrm{dom}(\rho)$ but $F^j \notin \mathrm{dom}(\rho)$ for the matcher ρ computed so far. If several different instantiations $\rho(F^{i_1}), \ldots, \rho(F^{i_k})$ are available we prefer the simplest one, cf. the heuristic for *choose_matcher*. An example for the usefulness of this heuristic is given in the next section.

8 Lemma Speculation by Reuse

Using the heuristics proposed in the previous section we can reuse the proof shell $PS = \langle \Phi_s, C_s \rangle$ from Fig. 1 for proving the step formulas $\psi_{i,s}$ of the conjectures ψ_i:[8]

φ	$\mathsf{plus}(\mathsf{sum}(x), \mathsf{sum}(y))$	$\equiv$	$\mathsf{sum}(\mathsf{app}(x, y))$	ψ_6
Φ	$F^1(G^1(x), G^2(y))$	$\equiv$	$G^3(H^1(x, y))$	
ψ	$\mathsf{plus}(\mathsf{len}(x), \mathsf{len}(y))$	$\equiv$	$\mathsf{len}(\mathsf{app}(x, y))$	–
ψ_0	$\times(\mathsf{prod}(x), \mathsf{prod}(y))$	$\equiv$	$\mathsf{prod}(\mathsf{app}(x, y))$	ψ_8
ψ_1	$\mathsf{plus}(\mathsf{dbl}(x), \mathsf{dbl}(y))$	$\equiv$	$\mathsf{dbl}(\mathsf{plus}(x, y))$	–
ψ_2	$\mathsf{app}(\mathsf{rev}(y), \mathsf{rev}(x))$	$\equiv$	$\mathsf{rev}(\mathsf{app}(x, y))$	ψ_7
ψ_3	$\mathsf{plus}(x, \mathsf{s}(y))$	$\equiv$	$\mathsf{s}(\mathsf{plus}(x, y))$	–
ψ_4	$\mathsf{plus}(\times(x, z), \times(y, z))$	$\equiv$	$\times(\mathsf{plus}(x, y), z)$	ψ_6
ψ_5	$\times(\mathsf{exp}(z, x), \mathsf{exp}(z, y))$	$\equiv$	$\mathsf{exp}(z, \mathsf{plus}(x, y))$	ψ_8
ψ_6	$\mathsf{plus}(x, \mathsf{plus}(y, z))$	$\equiv$	$\mathsf{plus}(\mathsf{plus}(x, y), z)$	–
ψ_7	$\mathsf{app}(x, \mathsf{app}(y, z))$	$\equiv$	$\mathsf{app}(\mathsf{app}(x, y), z)$	–
ψ_8	$\times(x, \times(y, z))$	$\equiv$	$\times(\times(x, y), z)$	ψ_4
ψ_9	$\mathsf{plus}(\times(x, z), \times(y, z))$	$\equiv$	$\times(z, \mathsf{plus}(x, y))$	ψ_6

Table 1.

For each i a matcher π_i for $\psi_{i,s}$ and the schematic step formula Φ_s is selected, i.e. $\pi_i(\Phi_s) = \psi_{i,s}$. Then *solve_catch* is called with the partially instantiated schematic catch $C'_i := \pi_i(C_s)$, yielding some proof obligations E_i and a solution ρ_i for the free function variables.

Like π_s from Section 3, the matcher π_0 replaces only function variables with function symbols, e.g. $\pi_0(F^1) = \times$ and $\pi_0(D^1) = \mathsf{add}$, whereas π_1 with $\pi_1(D^1) = \mathsf{s}(w_2)$ makes use of functional terms. This allows to reuse the proof of φ, involving the data structure list for a conjecture concerning the data structure number. Similarly

[8] $\times$ multiplies two numbers, prod multiplies the numbers in a list, $\mathsf{dbl}(x)$ computes $2 \times x$, rev reverses a list, and $\mathsf{exp}(x, y)$ computes x^y.

$\pi_2(F^1) = \mathsf{app}(w_2, w_1)$ swaps the arguments of F^1 and $\pi_8(G^1) = w_1$ uses a projection to match the schematic step formula.

The applicability of a proof shell can be increased if we "freeze variables to constants", i.e. a (universally quantified) variable $z \in \mathcal{V}$ is regarded as a new constant $z \in \Sigma_0$. Thus e.g. conjecture ψ_4, i.e. the distributivity law for $\times$ and plus, matches the schematic conjecture Φ because now $\pi_4(G^{1,2,3}) = \times(w_1, z) \in \mathcal{T}(\Sigma, \mathcal{W})$ can be used for the matcher.[9] This yields the partially instantiated schematic catch $C_4' = \pi_4(C_s) =$

$$\left\{ \begin{array}{lcl} \times(\mathsf{s}(x), z) & \equiv & F^2(n, \times(x, z)) \\ \mathsf{plus}(\mathsf{s}(x), y) & \equiv & D^4(n, \mathsf{plus}(x, y)) \\ \times(D^4(n, x), z) & \equiv & F^3(n, \times(x, z)) \\ \mathsf{plus}(F^2(x, y), z) & \equiv & F^3(x, \mathsf{plus}(y, z)) \end{array} \right\}$$

and *solve_catch* computes the solution

$$\rho_4 = \{F^2/\mathsf{plus}(z, w_2),\ D^4/\mathsf{s}(w_2),\ F^3/\mathsf{plus}(z, w_2)\}.$$

In the same way the reuse is enabled for the remaining conjectures $\psi_5, \ldots, \psi_9$, where the reuse for ψ_8 (the associativity of $\times$) succeeds only when freezing y and z using the matcher $\pi_8 = \{F^1/\times(w_1, \times(y, z)),\ G^1/w_1,\ G^3/\times(w_1, z),\ H^1/\times(w_1, y),\ D^1/\mathsf{s}(w_2)\}$. For solving the catch C_9' the heuristic "instantiate F^3 like F^2" given at the end of the previous section is required.

Often one of the proof obligations in E_i is a lemma, indicated by the last column in the table above ("$-$" denotes that all members of E_i can be symbolically evaluated to tautologies of the form $t \equiv t$). Thus e.g. the associativity of plus, i.e. ψ_6, is computed as a proof obligation when proving $\psi_{4,s}$ by reuse, cf. the last equation of C_4' above.[10] If such a lemma is not already known in advance, it is *speculated* by instantiating a schematic equation with the matcher ρ_i obtained by solving the remaining part of the catch. In a direct proof of the step formula $\psi_{i,s}$ such a useful lemma has to be generated explicitly, cf. e.g. [Walsh, 1994] for examples.

Moreover, a lemma speculated by reuse is often a *generalization* of a lemma which is generated in a direct proof. Thus e.g. a direct proof of $\psi_{4,s}$ would generate the lemma $\sigma(\psi_6) = \mathsf{plus}(z, \mathsf{plus}(\times(x, z), \times(y, z))) \equiv$
$$\mathsf{plus}(\mathsf{plus}(z, \times(x, z)), \times(y, z))$$
(where $\sigma = \{x/z,\ y/\times(x, z),\ z/\times(y, z)\}$) if we proceed by using axioms for modifying the induction conclusion until the induction hypothesis is applicable. For obtaining a proof of $\psi_{4,s}$, this lemma is usually generalized by the *inverse substitution rule*, cf. [Walther, 1994], yielding ψ_6. This generalization effort is saved by our approach as ψ_6 is the lemma which is speculated for $\psi_{4,s}$ by reuse.

9 Conclusion and Future Work

We have developed an heuristically controlled algorithm for solving the free variables of a partially instantiated schematic proof catch. This algorithm, which is implemented in our prototype of a learning prover, the PLAGIATOR-system [Brauburger, 1994], has proved successful for many examples, including those from Table 1. Hence we are able to verify these conjectures by *automatically* reusing the proofs of previously proved, similar conjectures. As a side effect useful lemmata are speculated by our method.

Table 1 also suggests a recursive organization of the reuse procedure as the proof obligations returned by our solution algorithm may also be proved by reuse. The (heuristic) control of this recursion for avoiding non-termination by cyclic reuses is subject to future work.

Another future topic is concerned with the management of learned schematic proofs for an efficient selection of the proof shell which is to be reused for a given, new conjecture. For the subtask of choosing a matcher between the schematic and the new conjecture we may adapt the above heuristics for rating second-order matchers.

[9] This kind of skolemisation is sound because for a skolem constant z and a formula $\psi[z]$ with a free variable z holds: $\forall z.\psi[z]$ is valid iff $\psi[z]$ is valid.

[10] The proof obligation obtained for the conjecture $\psi_{2,s}$ is an instance of the associativity of app, viz. the lemma $\sigma_2(\psi_7)$ where $\sigma_2 = \{z/\mathsf{add}(z, \mathsf{empty})\}$.

Acknowledgements. We thank J. Brauburger, S. Gerberding, J. Giesl and M. Protzen for comments and discussions.

References

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

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

[Giunchiglia and Walsh, 1992] Fausto Giunchiglia and Toby Walsh. A Theory of Abstraction. *Artificial Intelligence,* 57:323-389, 1992.

[Goldfarb, 198l] Warren D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science,* 13:225-230, 1981.

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

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

[Kolbe and Walther, 1994] Thomas Kolbe and Christoph Walther. Reusing Proofs. *Proceedings of the 11th European Conference on Artificial Intelligence,* Amsterdam, pp. 80-84. John Wiley & Sons, Ltd., 1994.

[Kolbe and Walther, 1995] Thomas Kolbe and Christoph Walther. Patching Proofs for Reuse. *Proc. European Conf. on Machine Learning,* Heraklion, pp. 303 - 306, 1995.

[Protzen, 1992] Martin Protzen. Disproving Conjectures. In *Proceedings of the 11th International Conference on Automated Deduction,* Saratoga Springs, pp. 340-354, 1992.

[Walsh, 1994] Toby Walsh. A Divergence Critic. In *Proceedings of the 12th International Conference on Automated Deduction,* Nancy, France, pp. 14-28. 1994.

[Walther, 1994] Christoph Walther. Mathematical Induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 2, pp. 127-227. Oxford University Press, 1994.