Many-Sorted Logic in a Learning Theorem Prover

Thomas Kolbe[1] **Sabine Glesner**[2]

[1] FB Informatik, TH Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Germany
[2] Fakultät für Informatik, Universität Karlsruhe,
Vincenz-Prießnitz-Straße 3, 76128 Karlsruhe, Germany
kolbe@informatik.th-darmstadt.de glesner@ipd.info.uni-karlsruhe.de

Abstract. In a learning theorem prover, formulas can be verified by reusing proofs of previously verified conjectures. Reuse proceeds by transforming a successful proof into a valid schematic formula which can be instantiated subsequently. In this paper, we show how this reuse approach is extended to many-sorted logic: We first present the logical foundations for reasoning w.r.t. different sortings. Then their operational realization is given by developing a many-sorted proof analysis calculus for extracting the sort constraints imposed by a proof. For guaranteeing the validity of subsequent instantiations, we extend the second-order matching calculi for retrieving and adapting schematic formulas such that the computed sort constraints are satisfied. Finally we demonstrate the relevance of our extensions with several examples of many-sorted reuse.

1 Introduction

The improvement of theorem provers by machine learning techniques has recently been realized successfully in a number of applications, cf. e.g. [2, 3, 10]. The PLAGIATOR-system [9] is a learning theorem prover based on the *reuse* of previously computed proofs by the method of *Kolbe & Walther* [7, 8]: From an abstract point of view, a given proof is transformed into a *valid* formula, which is generalized and instantiated subsequently by certain (second-order) substitutions while preserving its validity. More precisely, a given proof $AX \vdash \varphi$ of a conjecture φ from some axioms AX is analyzed and generalized, yielding a valid formula $C \rightarrow \Phi$ containing function variables instead of function symbols. Now for each new conjecture ψ where some (second-order) matcher π is found such that $\pi(\Phi) = \psi$, the original proof can be reused obtaining a set $\pi(C)$ of proof obligations for ψ, i.e. $\pi(C) \rightarrow \psi$ is valid and the reuse succeeds if $\pi(C)$ is verified.

However, problems arise if this reuse approach for unsorted logic is extended to *many-sorted logic*, where objects of different basic data structures like numbers, lists, trees etc. can be distinguished syntactically by specifying their *sort*, cf. e.g. [4]. A many-sorted logic is an unsorted logic parameterized by a *sorting*, i.e. a mapping which provides the sort information for variables and function symbols. This is commonly used in automated reasoning since more efficient calculi can be built which exploit the given sort information (here we do not consider more general *order-sorted* logics with hierarchical sort relations, cf. [11]).

As terms and formulas are interpreted w.r.t. the sorting of variables and function symbols occurring in them, also the *validity* of formulas depends on the specified sorting: Consider e.g. the formula $\phi := \ (\forall x \ x \equiv \mathsf{a}) \rightarrow \mathsf{b}_1 \equiv \mathsf{b}_2$ where $\mathsf{a}, \mathsf{b}_1, \mathsf{b}_2$ are constants and x is a variable. In unsorted logic or in a many-sorted logic where $\mathsf{a}, \mathsf{b}_1, \mathsf{b}_2$ and x have the same sort, ϕ is valid, while in a many-sorted logic where e.g. a, x have the sort A and $\mathsf{b}_1, \mathsf{b}_2$ have the different sort B, ϕ is not valid. Hence a successful proof of ϕ w.r.t. the first sorting cannot be reused without considering the sort information, because otherwise an obviously incorrect "proof" of ϕ w.r.t. the second sorting would be obtained. A simple remedy for this problem would be to admit only those substitutions for reuse where the sorting is exactly retained, but this approach is far too restrictive.

For obtaining a more general criterion concerning admissible substitutions, we must be able to *abstract* from the fixed sorting used in the proof of some conjecture φ: We extract the *sort constraints* a proof imposes on the symbols occurring in it, e.g. the constraint that the (range) sort of $\mathsf{b}_1, \mathsf{b}_2$ must be identical to the sort of x in our example above. In this way we obtain the general statement that φ is valid w.r.t. *each* sorting *satisfying* the sort constraints, and therefore the given proof of φ can be reused for verifying some conjecture ψ which is specified w.r.t. some *different* sorting only if the sort constraints are satisfied.[1]

In Section 2 we introduce some formal concepts and show that the validity of formulas w.r.t. sortings or sort constraints is retained when applying sorted (second-order) substitutions. Section 3 shows how a successful proof is analyzed yielding a set of sort constraints whose satisfaction guarantees the validity of instantiations. We further extend the notion of *proof shells* [8], which represent reusable proofs, by a component obtained by generalizing the sort constraints. In Section 4 we deal with the goal-directed instantiation of proof shells for new conjectures respecting the sort constraints and give examples for many-sorted reuse. We summarize in Section 5 and comment on implementational issues.

2 Many-Sorted Logic

We introduce the syntax and semantics of many-sorted logic as we use it throughout this paper. In contrast to common formalizations [4] which assign a priori sorts to (variable and function) symbols, we introduce *sortings* as special syntactic objects for assigning sorts to symbols. This allows us to reason about formulas w.r.t. different sortings without changing the formulas themselves.

2.1 Syntax of Many-Sorted Logic

The many-sorted language is built from the set $\mathcal{X}$ of first-order variables and the set $\Theta = \bigcup_n \Theta_n$ of function symbols which is the union of all function symbols of arity $n \in \mathbb{N}$. The set $\mathcal{T}(\Theta, \mathcal{X})$ of terms and the set $\mathcal{F}(\Theta, \mathcal{X})$ of formulas are built as usual, where only equations $t_1 \equiv t_2$ with $t_1, t_2 \in \mathcal{T}(\Theta, \mathcal{X})$ are used as predicates. For representing second-order substitutions, we introduce a second

[1] The usual *relativization* (using unary predicate symbols for each sort) to transform many-sorted into unsorted formulas is not helpful for our application, since we have to reason about the validity of some (unchanged) formula w.r.t. different sortings.

set $\mathcal{W} = \{w_i \mid i \in \mathbb{N}\}$ of *parameter* variables. Here $w_i \in \mathcal{W}_n = \{w_1, \ldots, w_n\}$ denotes the ith argument position of an n-ary function, i.e. a functional term $t \in \mathcal{T}(\Theta, \mathcal{W}_n)$ built from function symbols and parameter variables corresponds to the λ-term $\lambda w_1, \ldots, w_n.t$ from the λ-calculus. To obtain a many-sorted logic, we introduce *sortings* for assigning sorts to variables and function symbols:

Definition 1 (sorts, sort variables, sortings). Let $\mathcal{S}$ be a set of *sort symbols* and let $\mathcal{S}_{\Theta,\mathcal{X}} = \mathcal{S}_\Theta \cup \mathcal{S}_\mathcal{X}$ be the set of *sort variables*, where $\mathcal{S}_\mathcal{X} = \{S(x) \mid x \in \mathcal{X}\}$ and $\mathcal{S}_\Theta = \bigcup_n \mathcal{S}_{\Theta_n}$ with $\mathcal{S}_{\Theta_n} = \{S(f, i) \mid f \in \Theta_n, 0 \le i \le n\}$. A sorting δ is a function $\delta : \mathcal{S}_{\Theta,\mathcal{X}} \to \mathcal{S}$ from sort variables to sort symbols.

Compared to the usual notion of an $\mathcal{S}$*-ranked alphabet* [4], sort variables provide an indirection when assigning sorts: For a fixed sorting δ therefore $\delta(S(f, 0))$ denotes the range sort of a function symbol f and $\delta(S(f, i)), 1 \le i \le n$, denote the domain sorts of f. This formalization is better suited for dealing with sort constraints subsequently but makes no difference when defining well-sortedness:

The set of all (δ-)sorted terms of sort s consists of variables $x \in \mathcal{X}$ where $\delta(S(x)) = s$ and terms $f(t_1, \ldots, t_n)$ where $f \in \Theta_n$, $\delta(S(f, 0)) = s$, and t_i is a δ-sorted term of sort $\delta(S(f, i))$, for $1 \le i \le n$. Similarly δ-sorted formulas are built from δ-sorted equations $t_1 \equiv t_2$ where t_1, t_2 are δ-sorted terms of the same sort. Since all sort information is supplied by the sorting δ, we do not need any sort information in formulas, i.e. we use (unsorted looking) quantifiers like $\forall x$ instead of denoting the sort of a quantified variable by $\forall x : s$ for $\delta(S(x)) = s$.

In general, a substitution ξ is a partial function $\xi : \mathcal{X} \cup \Theta \to \mathcal{T}(\Theta, \mathcal{X} \cup \mathcal{W})$ whose finite domain is denoted by $\mathrm{dom}(\xi) \subseteq \mathcal{X} \cup \Theta$. A substitution $\sigma : \mathcal{X} \to \mathcal{T}(\Theta, \mathcal{X})$ is called a *first-order substitution.* An injective first-order substitution $\gamma : \mathcal{X} \to \mathcal{X}$ is a *variable renaming.* A *second-order substitution* π is a substitution $\pi : \Theta \to \mathcal{T}(\Theta, \mathcal{W})$ such that $\pi(f) \in \mathcal{T}(\Theta, \mathcal{W}_n)$ for each $f \in \Theta_n \cap \mathrm{dom}(\pi)$. We neither admit non-parameter variables from $\mathcal{X}$ occurring in $\pi(f)$ ("π is *closed*") nor variables from $\mathcal{X} \cup \mathcal{W}$ occurring in $\mathrm{dom}(\pi)$ ("π is *pure*").

First-order substitutions are applied to terms as usual. Variable renamings can also be applied to formulas, replacing variables in the scope of quantifiers. A second-order substitution π is applied to terms by $\pi(x) = x$ and $\pi(f(t_1, ..., t_n)) = \sigma(\pi(f))$ for the first-order substitution $\sigma = \{w_1/\pi(t_1), ..., w_1/\pi(t_1)\}$ on parameter variables. Applying π to a formula ϕ is done by preserving the structure of ϕ and replacing the terms contained in ϕ as described. The restrictions "closed" and "pure" for second-order substitutions prevent variables from being caught within the scope of quantifiers and preserve the closeness of formulas in applications. E.g. applying the non-closed substitution $\{\mathsf{b}/x\}$ to the closed formula $(\forall x\, \mathsf{f}(x) \equiv \mathsf{b}) \to \mathsf{f}(\mathsf{a}) \equiv \mathsf{b}$ yields the non-closed formula $(\forall x\, \mathsf{f}(x) \equiv x) \to \mathsf{f}(\mathsf{a}) \equiv x$.

As usual, a first-order substitution σ is δ-sorted if $\sigma(x)$ is a δ-sorted term of sort $\delta(S(x))$ for each $x \in \mathrm{dom}(\sigma)$. A second-order substitution π is δ-sorted if for each $f \in \Theta_n \cap \mathrm{dom}(\pi)$, the term $\pi(f)$ is δ_f-sorted of sort $\delta(S(f, 0))$ for the modified sorting $\delta_f := \{S(w_1) \mapsto \delta(S(f, 1)), \ldots, S(w_n) \mapsto \delta(S(f, n))\} \circ \delta$. It is easy to show (by structural induction) that the δ-sortedness of terms and formulas is retained when applying a first-order substitution σ, variable renaming γ, or second-order substitution π which is δ-sorted, cf. [5].

2.2 Semantics of Many-Sorted Logic

The semantics of a δ-sorted logic is given as usual [4]: A δ-sorted algebra $M = (U, I)$ is a pair of a universe $U = \bigcup_s U_s$, where $U_s \neq \emptyset$ is the universe of sort $s \in \mathcal{S}$, and an interpretation I, mapping each function symbol $f \in \Theta$ to a function f_I on U of the appropriate arity and respecting $\delta(S(f,i)), 0 \leq i \leq n$. A δ-sorted variable assignment $V : \mathcal{X} \rightarrow U$ maps variables to elements of the universe of the appropriate sort. A formula ϕ is called δ-satisfiable if an algebra M exists such that $(M, V) \models_\delta \phi$ for each variable assignment V, where $\models_\delta$ denotes the meaning w.r.t. fixed M and V. A formula ϕ is δ-valid, written $\models_\delta \phi$, if $M \models_\delta \phi$ for each δ-sorted algebra M.

If a δ-sorted variable renaming γ is applied to a δ-valid, closed formula ϕ, then $\gamma(\phi)$ is also δ-valid because the application of γ results in a bound renaming of quantified variables which is known to be validity preserving. More interesting is the application of a second-order substitution π to a δ-valid formula, which is validity-preserving due to the properties of π being pure and closed, cf. [5]:

Theorem 2 (δ-validity under second-order substitutions). *Let δ be a sorting and let ϕ be a δ-sorted, closed formula. If ϕ is δ-valid, then $\pi(\phi)$ is also δ-valid for each δ-sorted second-order substitution π.*

For reasoning about the validity of a formula w.r.t. different sortings, however, we now replace the *absolute* sorting from Theorem 2 by a *relative* sorting, for which only the satisfaction of some *sort constraints* is required:

Definition 3 (sort constraints, collision sets, satisfy, *col*-valid). A *(sort) collision set col* $\subseteq \mathcal{S}^2_{\Theta,\mathcal{X}}$ is a set of pairs of sort variables $(S_1, S_2) \in \mathcal{S}^2_{\Theta,\mathcal{X}}$, called *sort constraints.* A sorting δ *satisfies* a collision set *col* iff it satisfies each contained sort constraint $(S_1, S_2) \in col$ by $\delta(S_1) = \delta(S_2)$. A formula ϕ is called *col-valid* iff ϕ is δ-valid for each sorting δ where *col* is satisfied and ϕ is δ-sorted.

E.g. the sort constraint $(S(f,0), S(x))$ represents that the range sort of the function symbol f must be identical to the sort of the variable x, cf. Definition 1, but without committing this sort to a specific $s \in \mathcal{S}$. Hence due to Theorem 2, "ϕ is *col*-valid" is a stronger statement w.r.t. instantiations than "ϕ is δ-valid":

Corollary 4 (*col*-validity under second-order substitutions). *Let col be a collision set and ϕ a closed col-valid formula. If the sorting δ satisfies col and ϕ is δ-sorted, then $\pi(\phi)$ is δ-valid for each δ-sorted second-order substitution π.*

Note that the δ-validity of a formula ϕ is *independent* of the part of δ concerning (sort variables for) symbols *not* occurring in ϕ. Hence when considering the δ-validity of ϕ we may modify δ for those new symbols.

Example 5 (*col*-validity vs. δ-validity). *Let δ be a sorting where the constants $\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4$ and the variables x, y have sort A, the constants $\mathsf{b}, \mathsf{b}_1, \mathsf{b}_2$ and the variable u have sort B, and the constant c and the variable v have sort C. Consider the following formulas and assume that ϕ is known to be δ-valid.*

$$\begin{aligned} \phi \; &:= (\forall x\ x \equiv \mathsf{a}_3) \wedge (\forall y\ y \equiv \mathsf{a}_4) \rightarrow \mathsf{a}_1 \equiv \mathsf{a}_2 \\ \phi' \; &:= (\forall u\ u \equiv \mathsf{b}) \wedge (\forall v\ v \equiv \mathsf{c}) \rightarrow \mathsf{b}_1 \equiv \mathsf{b}_2. \end{aligned}$$

Now we cannot apply Theorem 2 for showing the δ-validity of ϕ', because the second-order substitution $\pi := \{\mathsf{a}_3/\mathsf{b},\ \mathsf{a}_4/\mathsf{c},\ \mathsf{a}_1/\mathsf{b}_1,\ \mathsf{a}_2/\mathsf{b}_2\}$ and the variable renaming $\gamma := \{x/u,\ y/v\}$ with $\pi(\gamma(\phi)) = \phi'$ are both not *δ-sorted. However, if we even know ϕ to be col-valid for the collision set $col := \{(S(x), S(\mathsf{a}_1, 0))\}$, we can apply Corollary 4 for the modified sorting δ' where $\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, x$ have sort B and a_4, y have sort C, because δ' satisfies col, and ϕ as well as π and γ are δ'-sorted. Thus the δ'-validity and in turn the δ-validity of ϕ' is implied.*

Example 5 illustrates that the notion of *col*-validity allows to abstract from the specific sorting concerning the originally proven formula ϕ. Thus our results on the validity of formulas under variable renamings and second-order substitutions which respect a given (absolute or relative) sorting serve as logical basis of extending our reuse procedure to many-sorted logic. However, we must find a way for showing the *col*-validity of formulas for a collision set *col* to be determined.

3 Preparing Proofs for Reuse

For making a proof in many-sorted logic reusable, our goal according to Corollary 4 is to extract the sort collisions *col* the proof imposes on the symbols occurring in it, i.e. to transform the proof into a *col*-valid formula ϕ. As demonstrated in Example 5, reasoning about instantiations is simplified if two disjoint languages are used for specifying the original formula and the one obtained by instantiations, where (first- and second-order) substitutions connect both levels.

Therefore we assume the set $\mathcal{X}$ of variables from Section 2 to be divided into two disjoint subsets $\mathcal{X} =: \mathcal{V} \cup \mathcal{U}$, and the same holds for function symbols $\Theta_n =: \Sigma_n \cup \Omega_n$, $n \in \mathbb{N}$. Then $\mathcal{F}(\Sigma, \mathcal{V})$ denotes the set of formulas built from $\mathcal{V}$ and $\Sigma := \bigcup_n \Sigma_n$ which is used for expressing *specific* formulas, and $\mathcal{F}(\Omega, \mathcal{U})$ built from $\mathcal{U}$ and the set $\Omega := \bigcup_n \Omega_n$ of *function variables* is used for expressing *schematic* formulas.[2] The set of parameter variables $\mathcal{W}$ remains unchanged. We often use (partial) sortings $\delta_1 : \mathcal{S}_{\Sigma,\mathcal{V}} \to \mathcal{S}$ and $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \to \mathcal{S}$, where $\delta := \delta_1 \circ \delta_2$ denotes the (total) sorting with $\delta(S) := \delta_1(S)$ for $S \in \mathcal{S}_{\Sigma,\mathcal{V}}$ and $\delta(S) := \delta_2(S)$ for $S \in \mathcal{S}_{\Omega,\mathcal{U}}$. We let $\mathcal{V}(\phi)$ denote the variables from $\mathcal{V}$ occurring in ϕ etc.

Now we proceed as follows: We first extend the *proof analysis calculus* from [7] by a component for collecting sort constraints, such that a *col*-valid specific formula $\phi \in \mathcal{F}(\Sigma, \mathcal{V})$ is obtained from a proof. Then ϕ and *col* are *generalized* by mapping them to a schematic *Col*-valid formula $\Phi \in \mathcal{F}(\Omega, \mathcal{U})$ with $Col \subseteq \mathcal{S}^2_{\Omega,\mathcal{U}}$ which is stored in a *proof shell*, a data structure for representing reusable proofs [8]. Thus new, valid, specific formulas $\phi' = \pi(\gamma(\Phi)) \in \mathcal{F}(\Sigma, \mathcal{V})$ can be obtained by re-instantiating proof shells with substitutions respecting *Col*.

3.1 Many-Sorted Proof Analysis

In this subsection only formulas $\mathcal{F}(\Sigma, \mathcal{V})$ and sortings $\delta : \mathcal{S}_{\Sigma,\mathcal{V}} \to \mathcal{S}$ are used. We let x^* denote a tuple of variables, $\varphi|_o$ denotes the subterm of φ at position o, and $\varphi[o \leftarrow t]$ denotes subterm replacement at position o.

[2] We do *not* perform second-order reasoning by quantifying function variables etc.

In [7] a proof is modeled as a derivation in a simple proof calculus $\vdash_{AX}$, where deriving $\varphi \vdash_{AX}$ TRUE entails that the (conditional) equation φ is provable from the (equational) axioms AX using equational reasoning, i.e. an axiom $\forall x^* l \equiv r \in AX$ can be used for deriving $\varphi[o \leftarrow \sigma(r)]$ from φ if $\varphi|_o = \sigma(l)$ for some (first-order) substitution σ and some position o in φ. In [7], $\vdash_{AX}$ is extended to a proof *analysis* calculus $\vdash^{\mathsf{a}}_{AX}$ by collecting the applied axioms in an *accumulator* component A, i.e. deriving $\langle \varphi,\ \emptyset \rangle \vdash^{\mathsf{a}}_{AX} \langle \text{TRUE},\ A \rangle$ entails that also $\varphi \vdash_A$ TRUE can be derived and therefore $\models A \rightarrow \varphi$ holds for $A \subseteq AX$ (we use a set of formulas A also as a single formula: the conjunction of the elements of A).[3]

Since (equational) reasoning in many-sorted logic is done like in unsorted logic provided that all objects in the derivation are well-sorted, we can use the unsorted analysis calculus also for many-sorted proofs w.r.t. a fixed sorting δ [5]:

Lemma 6 (proof analysis with fixed sorting). *Let δ be a sorting, let AX be a set of axioms, let φ be a δ-sorted formula, and let $A \subseteq AX$ be δ-sorted. If $\langle \varphi,\ \emptyset \rangle \vdash^{\mathsf{a}}_{AX} \langle \text{TRUE},\ A \rangle$ is derived in the unsorted proof analysis calculus and each substitution σ used in this derivation is δ-sorted, then $\models_\delta A \rightarrow \varphi$.*

Lemma 6 demands that (rather obviously) the input φ and the output A of a derivation $\langle \varphi,\ \emptyset \rangle \vdash^{\mathsf{a}}_{AX} \langle \text{TRUE},\ A \rangle$ must be δ-sorted for guaranteeing $\models_\delta A \rightarrow \varphi$. Resuming Example 5, we show that the additional requirement concerning the δ-sortedness of applied substitutions is indeed necessary:

Example 7 (proof analysis and sorts). *The conjecture $\varphi := \mathsf{a}_1 \equiv \mathsf{a}_2$ can be verified from the axioms $AX = \{\forall x\ x \equiv \mathsf{a}_3,\ \forall y\ y \equiv \mathsf{a}_4\}$ in the simple proof analysis calculus obtaining the accumulator $A = \{\forall x\ x \equiv \mathsf{a}_3\}$, i.e. the formula $\phi := A \rightarrow \varphi$ is valid w.r.t. the sorting δ from Example 5:*

$\mathsf{a}_1 \equiv \mathsf{a}_2$	*apply axiom $\forall x\ x \equiv \mathsf{a}_3$ to a_1 with $\sigma_1 = \{x/a_1\}$*
$\mathsf{a}_3 \equiv \mathsf{a}_2$	*apply axiom $\forall x\ x \equiv \mathsf{a}_3$ to a_2 with $\sigma_2 = \{x/a_2\}$*
$\mathsf{a}_3 \equiv \mathsf{a}_3$	*built-in reflexivity of $\equiv$ yields* TRUE

But regarding this proof for verifying the same formula ϕ w.r.t. a new sorting δ' where a_3, x have the sort A and $\mathsf{a}_1, \mathsf{a}_2$ have a different sort B would be invalid since ϕ is well-sorted but does not hold w.r.t. δ'. The substitutions σ_1 and σ_2 used in the proof are only well-sorted w.r.t. the original but not the new sorting.

The example reveals the need for inspecting a specific proof and extracting the sort constraints the proof imposes on the symbols occurring in it. We represent this information concerning the well-sortedness of applied substitutions by certain collision sets, cf. Definition 3, which depend on the replaced term l:

Definition 8 (collision set for substitutions). For a (first-order) substitution σ and a term l, the collision set $col(\sigma, l) \subseteq \mathcal{S}^2_{\Sigma,\mathcal{V}}$ for σ w.r.t. l is defined by $col(\sigma, l) := \{(S(l), tls(\sigma(l)))\}$ if $l \in \mathcal{V}$ and $col(\sigma, l) := \emptyset$ if $l \notin \mathcal{V}$.
Here the function $tls : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{S}_{\Sigma,\mathcal{V}}$ yields a designator for the *top level sort* of a term, where $tls(x) := S(x)$ for $x \in \mathcal{V}$ and $tls(f(\ldots)) := S(f, 0)$ for $f \in \Sigma$.

[3] The *refined* analysis calculus from [7] additionally distinguishes different occurrences of function symbols (thus increasing the reusability of proofs), and the extension to many-sorted logic is done in the same way as described here.

Now $\vdash^{\mathrm{a}}_{AX}$ is extended to a *many-sorted* proof analysis calculus $\vdash^{\mathrm{ac}}_{AX}$ by collecting the collision set for used substitutions in an additional component, i.e. derivations have the form $\langle \varphi,\ \emptyset,\ \emptyset \rangle \vdash^{\mathrm{ac}}_{AX} \langle \textsc{true},\ A,\ \mathit{col} \rangle$ with $\mathit{col} \subseteq \mathcal{S}^2_{\Sigma,\mathcal{V}}$. Here for each application $\varphi[o \leftarrow \sigma(r)]$ of an axiom $\forall x^* l \equiv r \in AX$ using a substitution σ with $\varphi|_o = \sigma(l)$, the sort constraints $\mathit{col}(\sigma, l)$ are added to the *col*-component. The following theorem proven in [5] states that these collected sort constraints are enough to guarantee the well-sortedness of applied substitutions (note that *col*-validity of some formula ϕ requires ϕ only to be δ-valid for sortings δ where ϕ is δ-sorted (and *col* is satisfied), cf. Definition 3):

Theorem 9 (many-sorted proof analysis). *Let AX be a set of axioms, let φ be a formula, let $A \subseteq AX$ be an accumulator, and let col be a collision set such that $\langle \varphi,\ \emptyset,\ \emptyset \rangle \vdash^{\mathrm{ac}}_{AX} \langle \textsc{true},\ A,\ \mathit{col} \rangle$ is a derivation in the many-sorted proof analysis calculus. Then the formula $A \to \varphi$ is col-valid.*

Theorem 9 shows how the calculus $\vdash^{\mathrm{ac}}_{AX}$ can be used for simultaneously *proving* a conjecture φ from given axioms AX and *analyzing* the constructed proof w.r.t. applied axioms and necessary sort constraints: If φ and AX are specified w.r.t. a fixed sorting δ_0, then $AX \models_{\delta_0} \varphi$ is verified if a derivation $\langle \varphi,\ \emptyset,\ \emptyset \rangle \vdash^{\mathrm{ac}}_{AX} \langle \textsc{true},\ A,\ \mathit{col} \rangle$ in the many-sorted proof analysis calculus can be established where all applied substitutions are δ_0-sorted. But additionally — by analyzing the proof — the more general statement $\models_{\delta} A \to \varphi$ is verified where δ may be *any* sorting such that $A \to \varphi$ is well-sorted and *col* is satisfied.[4] For instance the reuse attempt described in Example 7 is prohibited as the sort constraints $\{(S(x), S(\mathsf{a}_1, 0)), (S(x), S(\mathsf{a}_2, 0))\}$ which are collected for the substitutions used in the proof are not satisfied by the new sorting δ'.

3.2 Constructing Proof Shells

The improved analysis technique avoids invalid proof reuses when considering conjectures specified for new sortings. For achieving the separation into specific and schematic formulas mentioned in the beginning of this section, we let generalizations map between the signatures Σ and Ω resp. the variable sets $\mathcal{V}$ and $\mathcal{U}$ (schematic objects are denoted by capital symbols):

Definition 10 (generalization). A *generalization* $\mu \circ \beta$ is a substitution built from a second-order substitution $\mu : \Sigma \to \mathcal{T}(\Omega, \mathcal{W})$, replacing function symbols $f \in \Sigma_n$ by functional terms $\mu(f) = F(w_1, \ldots, w_n)$ for function variables $F \in \Omega_n$, and a variable renaming $\beta : \mathcal{V} \to \mathcal{U}$.

A generalization $\mu \circ \beta$ can also be applied to sort collision sets by defining $\mu(S(f,i)) = S(F,i)$ for $\mu(f) = F(w_1, \ldots, w_n)$ and $\beta(S(x)) = S(\beta(x))$. E.g. $\{\mathsf{a}_3/F,\ \mathsf{a}_1/G,\ \mathsf{a}_2/H\} \circ \{x/u\}$ generalizes the specific symbols from Example 7.

We characterize *proof shells* [8] as a data structure for representing the essentials of a proof $\langle \varphi,\ \emptyset,\ \emptyset \rangle \vdash^{\mathrm{ac}}_{AX} \langle \textsc{true},\ A,\ \mathit{col} \rangle$ in the schematic language $\mathcal{F}(\Omega, \mathcal{U})$, extended by a component for (generalized) sort constraints:

[4] Theorem 9 also holds for more general calculi containing unification rules etc. as e.g. used in the Plagiator-system [9] for treating arbitrary formulas, cf. Section 5.

Definition 11 (proof shells). A proof shell $PS = \langle \Phi,\ C,\ Col\rangle$ is built from a closed second-order formula $\Phi \in \mathcal{F}(\Omega,\mathcal{U})$ (also called *schematic conjecture*), a set of closed second-order formulas $C \subseteq \mathcal{F}(\Omega,\mathcal{U})$ (also called *schematic catch*), and a collision set $Col \subseteq \mathcal{S}^2_{\Omega,\mathcal{U}}$ such that $C \rightarrow \Phi$ is Col-valid.

A proof shell captures the "idea" of a successful proof, viz. that the schematic catch C entails the schematic conjecture Φ for all sortings satisfying Col. E.g. $PS = \langle G \equiv H,\ \{\forall u\ u \equiv F\},\ \{(S(u), S(G,0)),\ (S(u), S(H,0))\}\rangle$ is a proof shell constructed from many-sorted analysis of the proof of φ from Example 7, using the generalization from above.

Theorem 12 (construction of proof shells). *For a derivation* $\langle \varphi,\ \emptyset,\ \emptyset\rangle \vdash^{\mathrm{ac}}_{AX}$ $\langle \textsc{true},\ A,\ col\rangle$ *in the many-sorted proof analysis calculus,* $PS := \langle \Phi,\ C,\ Col\rangle :=$ $\langle \mu(\beta(\varphi)),\ \mu(\beta(A)),\ \mu(\beta(col))\rangle$ *is a proof shell, where* $\mu \circ \beta$ *is a generalization with* $\Sigma(A \cup \{\varphi\}) \subseteq \mathrm{dom}(\mu)$ *and* $\mathcal{V}(A \cup \{\varphi\}) \subseteq \mathrm{dom}(\beta)$.

Proof. Follows easily from Theorem 9 by showing that the Col-validity of $C \rightarrow \Phi$ is implied by the col-validity of $A \rightarrow \varphi$: For each sorting $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \rightarrow \mathcal{S}$ such that $C \rightarrow \Phi$ is δ_2-sorted and Col is δ_2-satisfied, we define a corresponding sorting $\delta_1 : \mathcal{S}_{\Sigma,\mathcal{V}} \rightarrow \mathcal{S}$ w.r.t. $\mu \circ \beta$ by stipulating $\delta_1(S(f,i)) := \delta_2(S(F,i))$ for $\mu(f) = F(w_1,\ldots,w_n)$ and $\delta_1(S(x)) := \delta_2(S(u))$ for $\beta(x) = u$. Then Corollary 4 is applicable for col, $A \rightarrow \varphi$, $\delta_1 \circ \delta_2$, and μ, yielding the δ_2-validity of $C \rightarrow \Phi$. $\square$

So far we have formalized how proof shells are constructed by analyzing and generalizing successfully computed proofs. Now we show how proof shells are re-instantiated for obtaining proofs of new conjectures.

4 Reusing Proofs

In the remainder of this paper we assume that the new conjectures ψ to be proven are δ_1-sorted w.r.t. a fixed sorting $\delta_1 : \mathcal{S}_{\Sigma,\mathcal{V}} \rightarrow \mathcal{S}$ for the language defined by $\Sigma \cup \mathcal{V}$. When considering proof reuse, δ_1 must be extended by a sorting $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \rightarrow \mathcal{S}$ for the proof shell PS, such that a total sorting $\delta = \delta_1 \circ \delta_2$ is obtained for checking the well-sortedness of substitutions and the sort constraints. To commit these language restrictions, we let "mapper" denote a second-order substitution $\Omega \rightarrow \mathcal{T}(\Sigma,\mathcal{W})$, "renaming" denotes a variable renaming $\mathcal{U} \rightarrow \mathcal{V}$, and "conjecture" denotes a closed δ_1-sorted first-order formula. The *goal-directed* computation of admissible mappers π and renamings γ for instantiating a proof shell w.r.t. given δ_1 and ψ (guaranteeing the existence of a suited sorting δ_2) is based on an algorithm for *sorted second-order matching* which is presented first. Finally we give examples of many-sorted reuse revealing the gains of our treatment of sorts, as naive approaches would restrict the reusability of proofs.

4.1 Many-Sorted Second-Order Term-Matching

An unsorted second-order matching problem $p \lhd t$ for a *pattern* $p \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V} \cup \mathcal{U})$ and a *target* $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is solved by computing a mapper $\pi : \Omega \rightarrow \mathcal{T}(\Sigma, \mathcal{W})$ with $\pi(p) = t$ (we perform "pure" second-order matching as first-order variables in the pattern are not instantiated). The standard algorithm from [6] uses the operations *decomposition*, *projection*, and *imitation* for solving a (generally simultaneous) second-order matching problem $R := [p_1 \lhd t_1, \ldots, p_n \lhd t_n]$. Since several operations may be applicable, branching leads to multiple solutions, and we let $\Pi := match(R)$ denote the set Π of mappers computed by this calculus.

For extending *match* w.r.t. sorts, we provide a flexible way to express the well-sortedness of objects by defining the well-sortedness of arbitrary collision sets $Q \subseteq \mathcal{S}^2_{\Sigma\cup\Omega,\mathcal{V}\cup\mathcal{U}}$, cf. Definition 3, w.r.t. a fixed sorting $\delta_1 : \mathcal{S}_{\Sigma,\mathcal{V}} \to \mathcal{S}$:[5]

Definition 13 (δ_1-sorted collision sets). A collision set $Q \subseteq \mathcal{S}^2_{\Sigma\cup\Omega,\mathcal{V}\cup\mathcal{U}}$ of sort constraints is δ_1*-sorted* iff $S_1 \sim_Q S_2$ for $S_1, S_2 \in \mathcal{S}_{\Sigma,\mathcal{V}}$ implies $\delta_1(S_1) = \delta_1(S_2)$, where $\sim_Q \subseteq \mathcal{S}^2_{\Sigma\cup\Omega,\mathcal{V}\cup\mathcal{U}}$ is the equivalence relation induced by Q.

Hence a collision set Q is δ_1-sorted iff there is some sorting $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \to \mathcal{S}$ such that $\delta_1 \circ \delta_2$ satisfies Q. For instance both collision sets $Q_1 := \{(S(F,1), S(k))\}$ and $Q_2 := \{(S(\mathsf{len},1), S(F,0)),\ (S(F,1), S(m)\}$ are δ_1-sorted if k resp. m is a variable of δ_1-sort list resp. nat and len computes the length of a list, but their union $Q_3 := Q_1 \cup Q_2$ is not δ_1-sorted because $S(k) \sim_{Q_3} S(m)$ but $\delta_1(S(k)) = \mathsf{list} \neq \mathsf{nat} = \delta_1(S(m))$. Now the well-sortedness (w.r.t. $delta_1$) of terms, formulas, and substitutions can be expressed by certain collision sets, viz. *contexts*:

Definition 14 (context of terms). The *context* $con(p)$ of a term $p \in \mathcal{T}(\Sigma \cup \Omega, \mathcal{V} \cup \mathcal{U})$ is the collision set inductively defined by (cf. Definition 8 for *tls*)

$$\begin{aligned} con(z) &:= \emptyset, && \text{if } z \in \mathcal{V} \cup \mathcal{U} \\ con(f(p_1,\ldots,p_n)) &:= \textstyle\bigcup_i \ \{(S(f,i),\ tls(p_i))\} \cup con(p_i), && \text{if } f \in \Sigma_n \cup \Omega_n. \end{aligned}$$

The context of a term represents all sort constraints which are implicitly given by the term's structure, i.e. $con(p)$ is δ_1-sorted iff there is some sorting $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \to \mathcal{S}$ such that p is $(\delta_1 \circ \delta_2)$-sorted. For terms $p_1, \ldots, p_n$ we have $\bigcup_i con(p_i)$ δ_1-sorted iff all p_i are $(\delta_1 \circ \delta_2)$-sorted w.r.t. some *same* sorting δ_2. E.g. the terms $p_1 := F(k)$ and $p_2 := \mathsf{len}(F(m))$ are not $(\delta_1 \circ \delta_2)$-sorted w.r.t. any sorting δ_2 because the union Q_3 of the above collision sets $Q_1 = con(p_1)$, $Q_2 = con(p_2)$ obtained as contexts is not δ_1-sorted. Similarly contexts are defined for formulas and substitutions, e.g. $con(\pi) = \{(S(F,0), S(\mathsf{len},0)),\ (S(\mathsf{len},1), S(G,0)),\ (S(G,1), S(F,2))\}$ for $\pi = \{F/\mathsf{len}(G(w_2))\}$, where the parameter variable w_2 points to $S(F,2)$.

We extend *match* to an algorithm *sorted_match*(R, Q) yielding the matchers π of R for which $Q \cup con(\pi)$ is δ_1-sorted for an initially given collision set Q, cf. [5]: *During the matching process*, Q is updated to Q' by adding the contexts of the stepwise constructed substitutions, where the actual branch is aborted if Q' is not δ_1-sorted and otherwise the branch is continued with $Q := Q'$. Thus parts of the search space are cut by early detecting violations of sort constraints.

[5] The algorithm from [6] already assumes that a fixed sort (called "(elementary) type") is given for all symbols, and the matching operations are extended there by conditions checking these sorts. Our sort constraints rather correspond to *polymorphic types* in the typed λ-calculus, e.g. the sort constraint $(S(F,1), S(F,0))$ for $F \in \Omega_2$ resembles the typing $F_{\alpha\times\beta\,\to\,\alpha}$ where α, β are *type variables* (which can be instantiated by types). We did not follow the way of extending [6] to polymorphic types as our notion of sort constraints allows more compact representations and efficient tests.

4.2 Retrieval and Adaptation of Proof Shells

We formulate our approach to many-sorted reuse using the notion of δ_1-sorted collision sets. Instantiating a proof shell $\langle \Phi,\ C,\ Col\rangle$ is split into two phases, resulting in a partially or totally instantiated catch, respectively: For *retrieval* the schematic conjecture Φ is matched with a new conjecture ψ, and during *adaptation* the axioms AX for ψ are used for instantiating the remaining symbols from the schematic catch C such that provable formulas are obtained.

Theorem 15 (reusing proofs by retrieval and adaptation). *If, for a conjecture ψ and a proof shell $PS = \langle \Phi,\ C,\ Col\rangle$, there are a mapper π and a renaming γ such that $\pi(\gamma(\Phi)) = \psi$ and $Q_p := Col \cup con(C \cup \{\Phi\}) \cup con(\pi) \cup con(\gamma)$ is δ_1-sorted, then we say PS* applies for *ψ (via $\pi \circ \gamma$) and we call $C_p := \pi(\gamma(C))$ the* partially instantiated catch. *If there further are a mapper ρ and a renaming η such that $C_t := \rho(\eta(C_p)) \subseteq \mathcal{F}(\Sigma, \mathcal{V})$ and $Q_t := Q_p \cup con(\rho) \cup con(\eta)$ is δ_1-sorted, then the* totally instantiated catch *C_t is δ_1-sorted and $\models_{\delta_1} C_t \rightarrow \psi$, and we say ψ* is reduced to *C_t (by PS via $\pi \circ \gamma \circ \rho \circ \eta$).*

Proof. Let π, γ and ρ, η be given as required. Then there is a sorting $\delta_2 : \mathcal{S}_{\Omega,\mathcal{U}} \rightarrow \mathcal{S}$ such that $C \rightarrow \Phi$, $\pi' := \pi \circ \rho$, and $\gamma' := \gamma \circ \eta$ are δ-sorted and δ satisfies Col, for the sorting $\delta := \delta_1 \circ \delta_2$. Therefore $\models_\delta \pi'(\gamma'(C)) \rightarrow \pi'(\gamma'(\Phi))$ is implied by Corollary 4 and Definition 11. Since $\pi(\gamma(\Phi)) = \psi$ implies $\pi'(\gamma'(\Phi)) = \psi$ and further $\pi'(\gamma'(C)) = C_t \subseteq \mathcal{F}(\Sigma, \mathcal{V})$ holds, we have even $\models_{\delta_1} C_t \rightarrow \psi$. □

To treat a *formula*-pair $\langle \Phi, \psi\rangle$ with the algorithm *sorted_match* for *terms*, $\langle R, \gamma\rangle := decompose(\Phi, \psi)$ denotes the preprocessing step of structurally comparing Φ and ψ (up to quantified variables and terms in equations). E.g. $R := [F(u) \lhd a(x),\ G(v) \lhd b,\ H(u,v) \lhd f(y),\ D \lhd c]$ and $\gamma := \{u/x,\ v/y\}$ results from decomposing $\forall u \forall v\ F(u) \equiv G(v) \wedge H(u,v) \equiv D$ and $\forall x \forall y\ a(x) \equiv b \wedge f(y) \equiv c$. Hence γ is a renaming if *decompose* succeeds, i.e. $PS = \langle \Phi,\ C,\ Col\rangle$ applies for ψ via $\pi \circ \gamma$ for each (if any) $\pi \in sorted\_match(\gamma(R), Q)$, if the collision set $Q := Col \cup con(C \cup \{\Phi\}) \cup con(\gamma)$ is δ_1-sorted, cf. Theorem 15. Here $con(C) \subseteq Q$ ensures the sort constraints imposed by the schematic catch C to be checked already during retrieval, i.e. some "mappers" with $\pi(\gamma(\Phi)) = \psi$ are excluded early because there is no δ_1-sorted *total* instantiation of C.

The obtained partially instantiated catch $C_p := \pi(\gamma(C))$ may still contain function variables, stemming from function symbols which appear in the original proof but not in the original conjecture. These *free* function variables are instantiated during the *adaptation* phase: An efficient procedure *solve_catch* incorporates the underlying axioms for ψ by heuristically combining a second-order matching algorithm with the technique of *symbolic evaluation*, cf. [8]. This immediately transfers to many-sorted reuse, where the obtained δ_1-sorted collision set Q_p serves as input for calls of *sorted_match* when further processing C_p.

Hence the presented reuse method reduces the provability of a new conjecture to the provability of a set of speculated conjectures, i.e. for a given underlying set of axioms AX, we have verified $AX \models_{\delta_1} \psi$ if we can show $AX \models_{\delta_1} C_t$. Since the remaining proof obligations C_t can again be proved by reuse, *recursion* is recommendable for the reuse procedure, cf. [8] for controlling termination.

4.3 Examples of Many-Sorted Reuse

We consider some examples from the viewpoint of many-sorted reuse, i.e. we analyze how our techniques for extracting, generalizing and instantiating sort constraints enable proof reuses which were excluded by naive approaches. In some examples we exploit that the applicability of proof shells is increased if one requirement of Theorem 15 is relaxed by demanding only $\pi(\gamma(\Phi)) \cong \psi$, where $\cong$ allows several equivalence preserving transformations. Transformations like swapping equations or reordering subformulas can be built into the calculus for matching formulas, cf. [1] for recent improvements. Our examples stem from the domain of theorem proving by mathematical induction, cf. Table 1: The proof shell computed from the given proof of the step formula for φ_0 (in the first row) is reused for proving the step formulas for the remaining statements φ_1, φ_2, etc.[6]

The last column shows how the soundness of the many-sorted reuse in the respective row is guaranteed. Here (a) denotes that in the proof by reuse for each symbol exactly the same sorts as in the original proof are used, and (b) denotes that in the proof by reuse only one overall sort is used, i.e. in these cases the soundness of the many-sorted reuse is obvious and our extensions are not necessary. But in the remaining cases (c) only our construction of sort constraints guarantees the validity of instantiations, because e.g. different function symbols in the original proof with the *same* (range and domain) sorts are mapped to different function symbols in the proof by reuse with *different* sorts, however respecting the computed sort constraints. This situation is repeated for other source proofs, i.e. our techniques count for a significant increase of reusability.

φ_0	$\sum k + \sum l \;\equiv\; \sum(k <> l)$	
No.	Conjectures proved by reuse	Sorting
φ_1	$\prod k \times \prod l \;\equiv\; \prod(k <> l)$	(a)
φ_2	$\vert k \vert + \vert l \vert \;\equiv\; \vert k <> l \vert$	(a)
φ_3	$m \times \mathsf{i} + n \times \mathsf{i} \;\equiv\; (m+n) \times \mathsf{i}$	(b)
φ_4	$m + (n + \mathsf{i}) \;\equiv\; (m+n) + \mathsf{i}$	(b)
φ_5	$\vert k <> l \vert \;\equiv\; \vert l <> k \vert$	(c)
φ_6	$\vert k <> n :: \mathsf{empty} \vert \;\equiv\; \mathsf{succ}(\vert k \vert)$	(c)
φ_7	$\vert k <> n :: \mathsf{l} \vert \;\equiv\; \mathsf{succ}(\vert k <> \mathsf{l} \vert)$	(c)
φ_8	$\mathsf{incr}(\mathsf{m}, k) <> \mathsf{incr}(\mathsf{m}, l) \;\equiv\; \mathsf{incr}(\mathsf{m}, k <> l)$	(c)
φ_9	$\mathsf{nthcut}(m, \mathsf{nthcut}(n, \mathsf{k})) \;\equiv\; \mathsf{nthcut}(m+n, \mathsf{k})$	(c)
φ_{10}	$\mathsf{reverse}(\mathsf{reverse}(k)) \;\equiv\; k$	(c)
φ_{11}	$\vert \mathsf{reverse}(k) \vert \;\equiv\; \vert k \vert$	(c)
φ_{12}	$\mathsf{reverse}(k <> n :: \mathsf{empty}) \;\equiv\; n :: \mathsf{reverse}(k)$	(c)
φ_{13}	$\mathsf{or}(\mathsf{member}(\mathsf{m}, k), \mathsf{member}(\mathsf{m}, l)) \;\equiv\; \mathsf{member}(\mathsf{m}, k <> l)$	(c)

Table 1. Conjectures proved by reusing the proof of φ_0

[6] The following functions operate on lists: $\sum$ sums up all elements, $<>$ denotes concatenation, $\prod$ multiplies all elements, $\vert\,.\,\vert$ yields the length, :: adds an element, incr increments each element, nthcut cuts elements from the back end, reverse reverses the order of elements, and member tests for occurrence of elements.

5 Conclusion

We have shown that a learning theorem prover specified for unsorted logic cannot be used for many-sorted logic without further extensions. Learning of proofs is based on their reuse, i.e. from a logical perspective a given proof is transformed into a *valid* formula which can be generalized and instantiated subsequently by certain substitutions while preserving its validity. For many-sorted reuse we have shown that it is necessary to also learn the sort information contained in a formula and its proof to ensure the soundness of instantiations. This allows us to abstract from a specific sorting and reason about the validity of (instantiated) formulas w.r.t. different sortings for the contained symbols.

It turned out that the learning theorem prover can be extended to many-sorted logic with moderate effort as the overall architecture of the reuse procedure remains unchanged. The described extensions for many-sorted logic are implemented in the PLAGIATOR-system [9], the prototype of a learning theorem prover which formerly performed unsorted reuse. The examples given here reveal that using the developed approach to many-sorted reuse increases the reusability of proofs compared to a naive treatment of sorts. An extension to order-sorted logic [11] by interpreting sort constraints as subsort-relations seems possible.

Acknowledgments: We would like to thank Jürgen Giesl and Wolf Zimmermann for many helpful discussions and comments on earlier drafts of this paper.

References

1. R. Curien, Z. Qian, and H. Shi. Efficient Second-Order Matching. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, pages 317 – 331, New Brunswick, NJ, USA, 1996. Springer LNCS 1103.
2. J. Denzinger and S. Schulz. Learning Domain Knowledge to Improve Theorem Proving. In *Proceedings CADE-13*, pages 62 – 76. Springer LNAI 1104, 1996.
3. M. Fuchs. Experiments in the Heuristic Use of Past Proof Experience. In *Proceedings CADE-13*, pages 523 – 537. Springer LNAI 1104, 1996.
4. J. H. Gallier. *Logic for Computer Science.* John Wiley & Sons, 1987.
5. S. Glesner. Many-Sorted Logic in a Learning Prover. Diploma Thesis (in German), TH Darmstadt, 1996.
6. G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11:31–55, 1978.
7. T. Kolbe and C. Walther. Reusing Proofs. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94),* Amsterdam, The Netherlands, pages 80–84. John Wiley & Sons, Ltd., 1994.
8. T. Kolbe and C. Walther. Termination of Theorem Proving by Reuse. In *Proceedings CADE-13*, pages 106 – 120. Springer LNAI 1104, 1996.
9. T. Kolbe and J. Brauburger. PLAGIATOR – A Learning Prover. In *Proceedings CADE-14.* Springer LNAI 1249, 1997.
10. E. Melis and J. Whittle. Internal Analogy in Theorem Proving. In *Proceedings CADE-13*, pages 92 – 105. Springer LNAI 1104, 1996.
11. M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer LNAI 395, 1989.