

Certifying and Synthesizing Membership Equational Proofs

Grigore Roşu², Steven Eker¹, Patrick Lincoln¹, and José Meseguer²

¹ Computer Science Laboratory
SRI International, USA

² Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Abstract. As the systems we have to specify and verify become larger and more complex, there is a mounting need to combine different tools and decision procedures to accomplish large proof tasks. The problem, then, is how to be sure that we can *trust* heterogeneous proofs produced by different tools based on different formalisms. In this work we focus on certification and synthesis of *equational proofs*, that are pervasive in most proof tasks and for which many tools are poorly equipped. Fortunately, equational proof engines like ELAN and Maude can perform millions of equational proof steps per second which, if properly certified, can be trusted by other tools. We present a general method to certify and synthesize proofs in membership equational logic, where the synthesis may involve generating full proofs from proof traces *modulo* combinations of associativity, commutativity, and identity axioms. We propose a simple representation for proof objects and give algorithms that can synthesize space-efficient, machine-checkable proof objects from proof traces.

1 Introduction

As the systems we have to specify and verify become larger and more complex, there is a mounting need to use specialized high-performance proof engines and efficient decision procedures, because general-purpose single-formalism approaches typically do not scale up well to large tasks [1]. For this reason, in many tools (e.g., [18, 17, 9]) such general formalisms are combined or extended with a variety of decision procedures and specialized proof accelerators. More generally, there is a growing interest in supporting *heterogeneous proofs*, in which not a single tool, but a combination of tools, developed by different researchers and based on different formalisms, can cooperate to solve an overall proof task. The problem, however, is how to be sure that we can *trust* the correctness of a heterogeneous proof. We have called this problem the *formal interoperability problem* [14] (see also [22, 15]). It can be naturally decomposed into two subproblems: (1) relating the semantics of different formalisms by adequate *maps of logics* [11, 7, 22, 15] that are then proved correct; and (2) providing machine-checkable *proof objects* for the proof subtasks carried out within each formalism by each tool. For example, in [20, 21] subproblems (1) and (2) have been solved in a special case by proving the correctness of a mapping between the logics of HOL [8] and NuPRL [6], and by defining a mapping between HOL and NuPRL proof objects.

In this paper we focus on subproblem (2), that is, on how to *certify* in a system-independent and machine-checkable way *proofs* carried out in a given formalism. This means that *we do not have to trust the tool* proving the subtask in question: if a machine-checkable proof of the result is provided, then

it can be certified using a trusted proof checker. More specifically, we focus on certification of *equality proofs*, which are typically represented as sequences of uses of the straightforward inference rules of equational logics. There is great practical interest in certified equality proofs not only because of the pervasive need for equational reasoning, but also because of the need to use fast equational engines within theorem proving tasks: on the one hand, some, otherwise quite powerful, theorem provers such as provers based on higher-order logics, need to carry out equality proofs step by step and do not scale up well to large equality proofs [1]; on the other hand, the wealth of results and techniques in equational reasoning and term rewriting, combined with advanced compilation technology, has made possible the recent development of very high performance *equational proof engines* such as ELAN [2] and Maude [5] that routinely carry out millions of equational proof steps per second. Furthermore, the equational proofs carried out by such engines are quite sophisticated: in ELAN they can be proofs *modulo* associativity and commutativity (AC), and in Maude they can be proofs *modulo* any combination of associativity (A), commutativity (C), and identity (U) axioms for different operators.

The power of equational deduction modulo such axioms brings about a corresponding *proof synthesis problem*: the tools must certify not only their *explicit* deduction steps, but also their *implicit* equational reasoning modulo axioms such as A, C, and U. For example, Maude provides a *trace* of equality steps that, in the simpler case of unconditional equations, looks roughly as follows:

$$t_0 \stackrel{e_1}{=} t_1 \stackrel{e_2}{=} t_2 \dots t_{n-1} \stackrel{e_n}{=} t_n$$

with e_i the equation used in the i^{th} step. However, since each such step is performed *modulo* axioms, say \mathcal{AX} , to have a full proof object certifying the task the implicit \mathcal{AX} -equality steps must be expanded out to:

$$t_0 \stackrel{\mathcal{AX}}{=} t'_0 \stackrel{e_1}{=} t_1 \stackrel{\mathcal{AX}}{=} t'_1 \stackrel{e_2}{=} t_2 \stackrel{\mathcal{AX}}{=} t'_2 \dots t_{n-1} \stackrel{\mathcal{AX}}{=} t'_{n-1} \stackrel{e_n}{=} t_n$$

where each proof $t_i \stackrel{\mathcal{AX}}{=} t'_i$ is no longer a one-step equality replacement, but may involve repeated application of the axioms in \mathcal{AX} . The proof synthesis problem involves synthesizing *short proofs* to fill the above “ \mathcal{AX} -gaps” $t_i \stackrel{\mathcal{AX}}{=} t'_i$. Since a typical trace may contain many millions of explicit equality steps, finding short proofs of the \mathcal{AX} -gaps is essential for scalability.

Our goals in this work are:

1. supporting the greatest possible generality in the kinds of equational proofs that can be certified and synthesized;
2. achieving the greatest possible simplicity, efficiency, and ease of check for the proof objects; and
3. synthesizing proofs that are as short as possible.

We address goal (1) by considering proofs in *membership equational logic*, a framework logic for equational reasoning [13] that contains many other logics (unsorted, many-sorted, order-sorted, etc.) as special cases, yet is efficiently implemented in Maude. Additional generality is gained by considering proofs *modulo* any combination of A, C, and U axioms. Our approach to goal (2) takes the form of extremely simple proof objects, that can be easily checked without

any need for parsing, and where proof subtasks can be *shared* for greater space and time checking efficiency. Goal (3) is addressed by new synthesis algorithms for \mathcal{AX} -gap proofs $t_i \stackrel{\mathcal{AX}}{=} t'_i$, with \mathcal{AX} any combination of A , C , and U axioms, that have length $\mathcal{O}(|t_i| \times \log(|t_i|))$ in the worse case.

Our work has been stimulated by recent work of Q.H. Nguyen, C. Kirchner, and H. Kirchner [16] on proof objects for equality proofs that can be used to certify to the Coq prover equality proofs carried out in the ELAN engine. Besides using a different proof representation that we think has important advantages, the main differences between our work and theirs are: (1) the greater generality of the equational logic (many-sorted in their case, membership equational logic in ours) and of the proofs modulo (AC in their case, any combination of A, C, and U in ours); and (2) the shorter length of the synthesized proofs for \mathcal{AX} -gap proofs ($\mathcal{O}(|t_i|^2)$ in their case, $\mathcal{O}(|t_i| \times \log(|t_i|))$ in ours).

2 Membership Equational Logic

In this section we recall membership equational logic (MEL) definitions and notations needed in the paper. The interested reader is referred to [13, 3] for a comprehensive exposition of MEL.

A *membership signature* Ω is a triple (K, Σ, π) where K is a set of *kinds*, Σ is a K -sorted (in this context called *K-kinded*) algebraic signature, and $\pi: S \rightarrow K$ is a function that assigns to each element in its domain, called a *sort*, a kind. Therefore, sorts are grouped according to kinds and operations are defined on kinds. For simplicity, we will call a “membership signature” just a “signature” whenever there is no confusion. For any given signature $\Omega = (K, \Sigma, \pi)$, an Ω -(*membership*) *algebra* A is a Σ -algebra together with a set $A_s \subseteq A_{\pi(s)}$ for each sort $s \in S$, and an Ω -homomorphism $h: A \rightarrow B$ is a Σ -homomorphism such that for each $s \in S$ we have $h_{\pi(s)}(A_s) \subseteq B_s$. Given a signature Ω and a K -indexed set of *variables*, an *atomic* (Ω, X) -*equation* has the form $t = t'$, where $t, t' \in T_{\Sigma, k}(X)$, and an *atomic* (Ω, X) -*membership* has the form $t : s$, where s is a sort and $t \in T_{\Sigma, \pi(s)}(X)$. An Ω -*sentence* in MEL has the form $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, where a, a_1, \dots, a_n are atomic (Ω, X) -equations or (Ω, X) -memberships, and $\{a_1, \dots, a_n\}$ is a set (no duplications). If $n = 0$, then the Ω -sentence is called *unconditional* and written $(\forall X) a$. Given an Ω -algebra A and a K -kinded map $\theta: X \rightarrow A$, then $A, \theta \models_{\Omega} t = t'$ iff $\theta(t) = \theta(t')$, and $A, \theta \models_{\Omega} t : s$ iff $\theta(t) \in A_s$. A *satisfies* $(\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, written $A \models_{\Omega} (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n$, iff for each $\theta: X \rightarrow A$, if $A, \theta \models_{\Omega} a_1$ and \dots and $A, \theta \models_{\Omega} a_n$, then $A, \theta \models_{\Omega} a$. An Ω -*specification* (or Ω -*theory*) $T = (\Omega, E)$ in MEL consists of a signature Ω and a set E of Ω -sentences. An Ω -algebra A *satisfies* (or *is a model of*) $T = (\Omega, E)$, written $A \models T$, iff it satisfies each sentence in E . We let \mathbf{MAlg}_T denote the full subcategory of \mathbf{MAlg}_{Ω} of membership Ω -algebras satisfying an Ω -theory T .

MEL admits complete deduction (see [13], where the rule of congruence is stated in a somewhat different but equivalent way; in the congruence rule below, $\sigma \in \Sigma_{k_1 \dots k_i, k}$, W is a set of variables $w_1 : k_1, \dots, w_{i-1} : k_{i-1}, w_{i+1} : k_{i+1}, \dots, w_n : k_n$, and $\sigma(W, t)$ is a shorthand for the term $\sigma(w_1, \dots, w_{i-1}, t, w_{i+1}, \dots, w_n)$):

- (1) Reflexivity : $\frac{}{E \vdash_{\Omega} (\forall X) t = t}$
- (2) Symmetry : $\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X) t' = t}$
- (3) Transitivity : $\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t' = t''}{E \vdash_{\Omega} (\forall X) t = t''}$
- (4) Congruence : $\frac{E \vdash_{\Omega} (\forall X) t = t'}{E \vdash_{\Omega} (\forall X, W) \sigma(W, t) = \sigma(W, t'), \text{ for each } \sigma \in \Sigma}$
- (5) Membership : $\frac{E \vdash_{\Omega} (\forall X) t = t', E \vdash_{\Omega} (\forall X) t : s}{E \vdash_{\Omega} (\forall X) t' : s}$
- (6) Modus Ponens : $\frac{\left\{ \begin{array}{l} \text{Given a sentence in } E \\ (\forall Y) t = t' \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m \\ \text{(resp. } (\forall Y) t : s \text{ if } t_1 = t'_1 \wedge \dots \wedge t_n = t'_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m) \\ \text{and } \theta : Y \rightarrow T_{\Sigma}(X) \text{ s.t. for all } i \in \{1, \dots, n\} \text{ and } j \in \{1, \dots, m\} \\ E \vdash_{\Omega} (\forall X) \theta(t_i) = \theta(t'_i), E \vdash_{\Omega} (\forall X) \theta(w_j) : s_j \end{array} \right.}{E \vdash_{\Omega} (\forall X) \theta(t) = \theta(t') \quad \text{(resp. } E \vdash_{\Omega} (\forall X) \theta(t) : s)}$

The rules above can therefore prove any unconditional equation or membership that is true in all membership algebras satisfying E . [19] gives a four-rule categorical equational inference system which can handle directly conditional equations, but that proof system is not well explored yet and seems hard to formalize at the level of detail needed in this paper. In order to derive conditional statements, we will therefore consider the standard technique adapting the “deduction theorem” to equational logics, namely deriving the conclusion of the sentence after adding the condition as an axiom; in order for this procedure to be correct, the variables used in conclusion need to be first transformed into constants. All variables can be transformed into constants, so we only consider the following simplified rules:

- (7) Theorem of Constants : $\frac{E \vdash_{\Omega \cup X} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}{E \vdash_{\Omega} (\forall X) a \text{ if } a_1 \wedge \dots \wedge a_n}$
- (8) Implication Elimination : $\frac{E \cup \{a_1, \dots, a_n\} \vdash_{\Omega} (\forall \emptyset) a}{E \vdash_{\Omega} (\forall \emptyset) a \text{ if } a_1 \wedge \dots \wedge a_n}$

Theorem 1. (from [13]) *With the notation above, $E \models_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$ if and only if $E \vdash_{\Omega} (\forall X) a$ if $a_1 \wedge \dots \wedge a_n$. Moreover, any statement can be proved by first applying rule (7), then (8), and then a series of rules (1) to (6).*

Maude [5] is an executable specification language supporting membership equational logic and also rewriting logic [12]. To make specifications easier to read, and to emphasize that order-sorted specifications are a special case of MEL ones, the following syntactic sugar conventions are supported by Maude:

Subsorts. Given sorts s, s' with $\pi(s) = \pi(s') = k$, the declaration $s < s'$ is syntactic sugar for the conditional membership $(\forall x : k) x : s' \text{ if } x : s$.

Operations. If $\sigma \in \Omega_{k_1 \dots k_n, k}$ and $s_1, \dots, s_n, s \in S$ with $\pi(s_1) = k_1, \dots, \pi(s_n) = k_n, \pi(s) = k$, then the declaration $\sigma : s_1 \dots s_n \rightarrow s$ is syntactic sugar for $(\forall x_1 : k_1, \dots, x_n : k_n) \sigma(x_1, \dots, x_n) : s \text{ if } x_1 : s_1 \wedge \dots \wedge x_n : s_n$.

Variables. $(\forall x : s, X) a \text{ if } a_1 \wedge \dots \wedge a_n$ is syntactic sugar for the Ω -sentence $(\forall x : \pi(s), X) a \text{ if } a_1 \wedge \dots \wedge a_n \wedge x : s$. With this, the operation declaration $\sigma : s_1 \dots s_n \rightarrow s$ is equivalent to $(\forall x_1 : s_1, \dots, x_n : s_n) \sigma(x_1, \dots, x_n) : s$.

We next give two examples of MEL specifications in Maude.

Example 1. This example is inspired by group theory. Suppose that a MEL specification defines a kind k , a constant $e : \rightarrow k$, a unary operation $_^- : k \rightarrow k$, a binary operation $_ \star _ : k \times k \rightarrow k$, and the three equations $(\forall A : k) e \star A = A$, $(\forall A : k) A^- \star A = e$, and $(\forall A, B, C : k) A \star (B \star C) = (A \star B) \star C$. The right-inverse and right-identity properties can then be proved.

In Maude, kinds are declared only implicitly, via declarations of sorts and subsorts. More precisely, a kind is automatically defined for every connected component in the partial order on sorts. Maude also implements efficient algorithms for rewriting modulo any combination of A, C, and U; however, in order to make proper use of them, the user needs to declare associativity, commutativity and/or identity as operator attributes rather than equations. The MEL specification of groups can then be implemented in Maude as follows:

```
fmod GROUP is sort S .
  op _*_ : S S -> S [assoc] .   op _- : S -> S .   op e : -> S .
  var A : S .   eq e * A = A .   eq (A -) * A = e .
endfm
```

The above specification is not confluent when regarded as a rewriting system. However, one can still use Maude to prove group properties, such as the right inverse law by giving two variants of a common expression modulo associativity, **reduce** $((A -) \star (A -)) \star (A \star (A -))$ and **reduce** $(A -) \star (((A -) \star A) \star (A -))$, which reduce to $A \star (A -)$ and e , respectively.

Example 2. Let us now consider a MEL specification of graphs with nodes, edges and paths, which additionally involves sorts and conditional statements. There are two kinds, k_n and k_p , for node and path elements, respectively. k_n contains a sort *Node*, and k_p contains *Edge* and *Path*. Any edge is a path, so we add the “subsort” membership (we label axioms to refer to them later):

[EdgeIsPath] $(\forall P : k_p) P : \text{Path} \text{ if } P : \text{Edge}.$

Source and target operators are also defined, $s, t : k_p \rightarrow k_n$, noticing that they return proper nodes only for proper paths:

[SourceNode] $(\forall P : k_p) s(P) : \text{Node} \text{ if } P : \text{Path},$
 [TargetNode] $(\forall P : k_p) t(P) : \text{Node} \text{ if } P : \text{Path}.$

Paths can be concatenated using the operator $_ ; _ : k_p \times k_p \rightarrow k_p$, but a correct path can be obtained only under appropriate conditions:

[CorrectPath] $(\forall E, P : k_p) E ; P : \text{Path} \text{ if } E : \text{Edge} \wedge P : \text{Path} \wedge t(E) = s(P).$

The source and target of a path are defined also under appropriate conditions:

[PathSource] $(\forall P, Q : k_p) s(P ; Q) = s(P) \text{ if } P : \text{Path} \wedge Q : \text{Path} \wedge t(P) = s(Q),$
 [PathTarget] $(\forall P, Q : k_p) t(P ; Q) = t(Q) \text{ if } P : \text{Path} \wedge Q : \text{Path} \wedge t(P) = s(Q).$

Finally, we introduce the associativity of path composition:

[Assoc] $(\forall P, Q, R : k_p) (P ; Q) ; R = P ; (Q ; R) \text{ if } P : \text{Path} \wedge Q : \text{Path} \wedge R : \text{Path} \wedge t(P) = s(Q) \wedge t(Q) = s(R).$

These axioms are all declared in the following (sugared) Maude specification:

```
fmod GRAPH is sorts Node Edge Path .   subsort Edge < Path .
  ops s t : Path -> Node .             op _;_ : Path Path -> [Path] .
  vars P Q R : Path .   var E : Edge .
  cmb E ; P : Path if t(E) = s(P) .
  ceq s(P ; Q) = s(P) if t(P) = s(Q) .   ceq t(P ; Q) = t(Q) if t(P) = s(Q) .
  ceq (P ; Q) ; R = P ; (Q ; R) if t(P) = s(Q) /\ t(Q) = s(R) .
endfm
```

3 Formalizing Proofs

In order to be mechanically checked, MEL proofs have to be first formalized. In this section we present such a formalization, for which a proof certifier will be given in the next section. By Theorem 1, any MEL sentence can be proved by first applying the theorem of constants, followed by implication elimination, and then by a series of applications of rules (1)–(6). This is reflected in our formalization of proofs by considering that proof objects have the following structure:

$$\begin{aligned} \langle \text{Proof Object} \rangle ::= & \langle \text{Proof Goal: Desugared Specification and Sentence} \rangle \\ & \langle \text{Theorem of Constants} \rangle \langle \text{Implication Elimination} \rangle \\ & \langle \text{Ground Proof} \rangle \end{aligned}$$

The proof goal, containing a specification and a sentence, both in desugared form, can be just a name referring to a file containing it. Keeping it isolated from proofs is a desirable feature for certifying authorities, who essentially want to consider proofs as nothing but mechanically checkable correctness certificates for well defined, often public, verification tasks. We do not enforce any specific syntax for defining and referring to the specification and the sentence to prove. The theorem of constants rule just adds constants to specification's signature:

$$\langle \text{Theorem of Constants} \rangle ::= (\text{constants } (\langle \text{Constant} \rangle : \rightarrow \langle \text{Kind} \rangle)^*)$$

$\langle \text{Constant} \rangle$ and $\langle \text{Kind} \rangle$ are identifiers. The syntax of the implication elimination rule is similar, in the sense that one adds equations or memberships to the specification; these new axioms should have (unique) labels:

$$\langle \text{Implication Elimination} \rangle ::=$$

$$(\text{implication } ([\langle \text{AxLabel} \rangle] \text{ (eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle \mid \text{mb } \langle \text{Term} \rangle : \langle \text{Sort} \rangle))^*)$$

$\langle \text{Sort} \rangle$ is an identifier. $\langle \text{Term} \rangle$ is a list of characters that the proof checker will ensure that is a well formed, disambiguated prefix term over the signature of the specification. A ground proof is a nonempty sequence of proof steps

$$\langle \text{Ground Proof} \rangle ::= (\langle \text{Proof Step} \rangle)^+,$$

where a proof step applies one of the rules (1)–(6). Each proof step is supposed to have a unique label, which is an integer number; the proof checker will ensure that these labels are used in increasing order:

$$\langle \text{Proof Step} \rangle ::= (\langle \text{Label} \rangle \langle \text{Rule} \rangle)$$

$$\begin{aligned} \langle \text{Rule} \rangle ::= & \langle \text{Reflexivity} \rangle \mid \langle \text{Symmetry} \rangle \mid \langle \text{Transitivity} \rangle \mid \\ & \langle \text{Congruence} \rangle \mid \langle \text{Membership} \rangle \mid \langle \text{Modus-Ponens} \rangle \end{aligned}$$

$$\langle \text{Reflexivity} \rangle ::= \text{reflexivity eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle$$

$$\langle \text{Symmetry} \rangle ::= \text{symmetry eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle \text{ follows by } \langle \text{Label} \rangle$$

$$\begin{aligned} \langle \text{Transitivity} \rangle ::= & \text{transitivity eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle \\ & \text{follows by } \langle \text{Label} \rangle \langle \text{Label} \rangle \end{aligned}$$

$$\begin{aligned} \langle \text{Congruence} \rangle ::= & \text{congruence eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle \\ & \text{position } \langle \text{Integer} \rangle \text{ follows by } \langle \text{Label} \rangle \end{aligned}$$

$$\begin{aligned} \langle \text{Membership} \rangle ::= & \text{membership mb } \langle \text{Term} \rangle : \langle \text{Sort} \rangle \\ & \text{follows by } \langle \text{Label} \rangle \langle \text{Label} \rangle \end{aligned}$$

$$\begin{aligned} \langle \text{Modus-Ponens} \rangle ::= & \text{modus-ponens} \\ & (\text{eq } \langle \text{Term} \rangle = \langle \text{Term} \rangle \mid \text{mb } \langle \text{Term} \rangle : \langle \text{Sort} \rangle) \\ & \text{axiom } \langle \text{AxLabel} \rangle \text{ map } ((\text{Var}):\langle \text{Kind} \rangle \leftarrow \langle \text{Term} \rangle)^* \\ & [\text{follows by } (\langle \text{Label} \rangle)^*] \end{aligned}$$

Spaces and new lines should be read as white spaces in the formalized syntax of MEL rules above. The list of labels in a modus-ponens rule application represents the proofs of the instantiated conditions of the axiom $\langle AxLabel \rangle$; not needed if the axiom is unconditional. The keywords can be shortened and some other encoding conventions can be devised in practical implementations in order to reduce the size of proofs, but essentially the same amount of information is needed in order to simplify the proof checker as much as possible.

Example 3. Let the group axioms in Example 1 have the labels *leftId*, *leftInv*, and *assoc*, respectively. In fully disambiguated form, as our proof certifier defined in the next section expects its input, these equational axioms are written as

$$\begin{aligned} (\forall A : k) \text{ } \neg \star_{-kk,k}(e_{\lambda,k}, A) &= A, \\ (\forall A : k) \text{ } \neg \star_{-kk,k}(\neg_{k,k}(A), A) &= e_{\lambda,k}, \\ (\forall A : k, B : k, C : k) \text{ } \neg \star_{-kk,k}(A, \neg \star_{-kk,k}(B, C)) &= \neg \star_{-kk,k}(\neg \star_{-kk,k}(A, B), C). \end{aligned}$$

Suppose now that the goal is to prove the right identity property, namely the equation $(\forall A : k) \text{ } \neg \star_{-kk,k}(A, e_{\lambda,k}) = A$. Then the following is a proof according to the formalization described above:

```
(constants  $a_{\lambda,k} : \rightarrow k$ ) (implication)
(1 modus-ponens eq  $\neg \star_{-kk,k}(e, \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) = \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))$ 
  axiom leftId map  $A : k \leftarrow \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))$  )
(2 modus-ponens eq  $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})) = e$  axiom leftInv map  $A : k \leftarrow \neg_{k,k}(a_{\lambda,k})$  )
(3 symmetry eq  $e = \neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k}))$  follows by 2)
(4 congruence eq  $\neg \star_{-kk,k}(e, \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})))$  position 1 follows by 3)
(5 symmetry eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(e, \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})))$  follows by 4)
(6 transitivity eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))$  follows by 5 1)
(7 modus-ponens eq  $\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), a_{\lambda,k}) = e$  axiom leftInv map  $A : k \leftarrow a_{\lambda,k}$  )
(8 congruence eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), a_{\lambda,k}), \neg_{k,k}(a_{\lambda,k})) = \neg \star_{-kk,k}(e, \neg_{k,k}(a_{\lambda,k}))$ 
  position 1 follows by 7)
(9 modus-ponens eq  $\neg \star_{-kk,k}(e, \neg_{k,k}(a_{\lambda,k})) = \neg_{k,k}(a_{\lambda,k})$  axiom leftId map  $A : k \leftarrow \neg_{k,k}(a_{\lambda,k})$  )
(10 transitivity eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), a_{\lambda,k}), \neg_{k,k}(a_{\lambda,k})) = \neg_{k,k}(a_{\lambda,k})$  follows by 8 9)
(11 modus-ponens eq  $\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), a_{\lambda,k}), \neg_{k,k}(a_{\lambda,k}))$ 
  axiom assoc map  $A : k \leftarrow \neg_{k,k}(a_{\lambda,k})$   $B : k \leftarrow a_{\lambda,k}$   $C : k \leftarrow \neg_{k,k}(a_{\lambda,k})$  )
(12 transitivity eq  $\neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) = \neg_{k,k}(a_{\lambda,k})$  follows by 11 10)
(13 congruence eq  $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})))) =$ 
   $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k}))$  position 2 follows by 12)
(14 modus-ponens eq  $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k}), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})))$  axiom assoc map
   $A : k \leftarrow \neg_{k,k}(a_{\lambda,k})$   $B : k \leftarrow \neg_{k,k}(a_{\lambda,k})$   $C : k \leftarrow \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))$  )
(15 symmetry eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(\neg_{k,k}(a_{\lambda,k}), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))))$  follows by 14)
(16 transitivity eq  $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k}))) =$ 
   $\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k}))$  follows by 15 13)
(17 symmetry eq  $\neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})) =$ 
   $\neg \star_{-kk,k}(\neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k})), \neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})))$  follows by 6)
(18 transitivity
  eq  $\neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})) = \neg \star_{-kk,k}(\neg_{k,k}(\neg_{k,k}(a_{\lambda,k})), \neg_{k,k}(a_{\lambda,k}))$  follows by 17 16)
(19 transitivity eq  $\neg \star_{-kk,k}(a_{\lambda,k}, \neg_{k,k}(a_{\lambda,k})) = e$  follows by 18 2)
```

Example 4. Let us now consider the MEL specification of graphs in Example 2. In order to keep the notation simple, in this example we do not write symbols in their disambiguated form, as we did in the previous example. However, our proof

checker described in the next section requires each symbol to be disambiguated. We next give a formalized proof object for the following conditional equation:

$(\forall N_1, N_3 : k_n; E_1, E_2, E_3 : k_p) (E_1; E_2); (E_3; E_1) = E_1; ((E_2; E_3); E_1) \text{ if}$
 $N_1 : \text{Node} \wedge N_2 : \text{Node} \wedge N_3 : \text{Node} \wedge E_1 : \text{Edge} \wedge E_2 : \text{Edge} \wedge E_3 : \text{Edge} \wedge$
 $s(E_1) = N_1 \wedge s(E_2) = N_2 \wedge s(E_3) = N_3 \wedge t(E_1) = N_2 \wedge t(E_2) = N_3 \wedge t(E_3) = N_1.$

Despite its apparently intuitive simplicity, the proof of the above implication is quite tedious. This is because one actually has to prove *all* the conditions of a sentence before applying it via the modus-ponens rule. In a mechanically checkable proof this information is needed.

```
(constants n1 :→ k_n n2 :→ k_n n3 :→ k_n e1 :→ k_p e2 :→ k_p e3 :→ k_p)
(implication [imp1] mb n1 : Node [imp2] mb n2 : Node [imp3] mb n3 : Node [imp4] mb e1 : Edge
[imp5] mb e2 : Edge [imp6] mb e3 : Edge [imp7] eq s(e1) = n1 [imp8] eq s(e2) = n2
[imp9] eq s(e3) = n3 [imp10] eq t(e1) = n2 [imp11] eq t(e2) = n3 [imp12] eq t(e3) = n1)
(1 modus-ponens mb e1 : Edge axiom imp4 map)
(2 modus-ponens mb e1 : Path axiom EdgeIsPath map P : k_p <- e1 follows by 1)
(3 modus-ponens mb e2 : Edge axiom imp5 map)
(4 modus-ponens mb e2 : Path axiom EdgeIsPath map P : k_p <- e2 follows by 3)
(5 modus-ponens eq s(e1) = n1 axiom imp7 map)
(6 symmetry eq n1 = s(e1) follows by 5)
(7 modus-ponens eq t(e3) = n1 axiom imp12 map)
(8 transitivity eq t(e3) = s(e1) follows by 7 6)
(9 modus-ponens mb e3 : Edge axiom imp6 map)
(10 modus-ponens mb e3 : Path axiom EdgeIsPath map P : k_p <- e3 follows by 1)
(11 modus-ponens mb e3; e1 : Path axiom CorrectPath map E : k_p <- e3 P : k_p <- e1
follows by 9 2 8)
(12 modus-ponens eq s(e2) = n2 axiom imp8 map)
(13 symmetry eq n2 = s(e2) follows by 12)
(15 transitivity eq t(e1) = s(e2) follows by 14 13)
(16 modus-ponens eq t(e2) = n3 axiom imp11 map)
(17 modus-ponens eq s(e3; e1) = s(e3) axiom PathSource map P : k_p <- e3 Q : k_p <- e1
follows by 10 2 8)
(18 modus-ponens eq s(e3) = n3 axiom imp9 map)
(19 transitivity eq s(e3; e1) = n3 follows by 17 18)
(20 symmetry eq n3 = s(e3; e1) follows by 19)
(21 transitivity eq t(e2) = s(e3; e1) follows by 16 20)
(22 modus-ponens eq (e1; e2); (e3; e1) = e1; (e2; (e3; e1)) axiom Assoc
map P : k_p <- e1 Q : k_p <- e2 R : k_p <- e3; e1 follows by 2 4 11 15 21)
(23 transitivity eq t(e2) = s(e3) follows by 21 17)
(24 modus-ponens eq (e2; e3); e1 = e2; (e3; e1) axiom Assoc
map P : k_p <- e2 Q : k_p <- e3 R : k_p <- e1 follows by 4 10 2 23 8)
(25 symmetry eq e2; (e3; e1) = (e2; e3); e1 follows by 24)
(26 congruence eq e1; (e2; (e3; e1)) = e1; ((e2; e3); e1) position 2 follows by 25)
(27 transitivity eq (e1; e2); (e3; e1) = e1; ((e2; e3); e1) follows by 22 26)
```

4 Certifying Proofs

The major reason why the proof objects formalized in the previous section are so low level is because a certifying authority will typically want to use a proof checker which is as simple as possible, in order to be easily validated and therefore trusted. A proof checker for generic logical frameworks, taking as input both a "logic", including its inference rules, and a proof within that logic, is a nice idea but in our view too complex to be easily trusted. Instead, we opt for logic specific proof checkers, having the inference rules of the logic, six in our case, hardwired. Interestingly, with the level of detail in proofs described above, the corresponding

proof checker described in what follows does *not* even need parsing; it only needs to perform trivial checks in time linear on the size of the proofs. We expect its implementation to be quite trivial and short. For the rest of this section we assume a proof object satisfying the syntax in the previous section

```
(constants  $c_{\lambda,k_1}^1 \rightarrow k_1 \dots c_{\lambda,k_C}^C \rightarrow k_C$ )
(implication [AxLabel1] sentence1 ... [AxLabelI] sentenceI)
(1 proofStep1)
...
(N proofStepN)
```

where $C, I \geq 0$ and $N \geq 1$, and claiming to prove a goal $Spec \models \varphi$. Then a proof certifier algorithm works as follows:

Proof goal. It first checks whether the specification and the sentence to prove satisfy the required format. This can be done by first collecting all the kinds and sorts declared in $Spec$, then all the operation declarations, checking whether each is correctly disambiguated, and finally checking each sentence in $Spec$ as well as the sentence to be proved. For simplicity, we also require and check that each variable defined in a sentence is different from any other symbol in the signature of $Spec$ and different from any other variable defined in the same sentence. A sentence $(\forall v_1 : k_1, \dots, v_V : k_V) a \text{ if } a_1 \wedge \dots \wedge a_m$ is then checked in three steps:

1. check that k_1, \dots, k_V are defined in $Spec$;
2. add $v_{\lambda,k_1}^1 \rightarrow k_1, \dots, v_{\lambda,k_V}^V \rightarrow k_V$ as disambiguated constants to $Spec$, and let $Spec'$ be the new specification; let also a', a'_1, \dots, a'_m be the atoms a, a_1, \dots, a_m with each occurrence of a variable v_i replaced by the constant v_{λ,k_i}^i , for each $1 \leq i \leq V$ (this can be simply and safely done by word/token substitution, because of the conventions on variable names).
3. check each atom a', a'_1, \dots, a'_m as follows: if it is a membership of the form $t : s$, then check that s is a sort of the kind k of t (k can be easily found by looking at the result of the topmost operator of t , because t is disambiguated), and then call the procedure $check(Spec', t)$, which is explained below; if the atom is an equality $t = t'$ then first check that the kinds of t and t' coincide and then call $check(Spec', t)$ and $check(Spec', t')$.

The procedure $check(Spec', t)$ ensures that t is a ground well-formed term under the syntax of $Spec'$. Since t is disambiguated, this is quite a trivial task: if t has the form $\sigma_{k_1 k_2 \dots k_n, k}(t_1, t_2, \dots, t_n)$ then we first check that the kind of each t_i is k_i (again, by just looking at the result kind of the topmost operator of t_i) and then recursively call $check(Spec', t_i)$ for each $1 \leq i \leq n$. The specification and the sentence to be proved are now checked, so we can start checking the proof.

Constants. Check that each constant $c_{\lambda,k_1}^1, \dots, c_{\lambda,k_C}^C$ is different from any other symbol in $Spec$ and from the other constants. Also check that C equals the number of variables in φ and that k_1, \dots, k_C are the kinds of those variables in this order, respectively, and that they are all defined in $Spec$. Add these constants to $Spec$. Let θ map variable $v_i : k_i$ in φ to constant c_{λ,k_i}^i , for each $1 \leq i \leq C$.

Implication. Check that φ has I conditions and that each of **sentence1**, ..., **sentenceI** is the instance by θ of the corresponding condition of φ ; check also that $[AxLabel1], \dots [AxLabelI]$ are distinct and different from other labels in $Spec$. Add all the labeled sentences to $Spec$.

Proof step. A proof step can have one of the following six types: reflexivity, symmetry, transitivity, congruence, membership and modus-ponens. Each type is analyzed separately. For simplicity of the checker, a common requirement, which needs to be checked, is that the proof step labels are given in increasing order. We next perform a case analysis on the type of the proof step:

- (1) **reflexivity** `eq $t = t'$` . Check that t and t' are equal *as strings of characters* (to keep the checker simple) and also call the procedure `check(Spec, t)`.
- (2) **symmetry** `eq $t = t'$ follows by $\langle label \rangle$` . Check that $\langle label \rangle$ is smaller than current label and that the (previous) proof step with label $\langle label \rangle$ was equational and proved equation $u = u'$, where u equals t' and u' equals t as strings.
- (3) **transitivity** `eq $t = t'$ follows by $\langle label_1 \rangle \langle label_2 \rangle$` . Check that $\langle label_1 \rangle$ and $\langle label_2 \rangle$ are smaller than the current label, that they refer to proof steps which are equational and prove the equations $u = u'$ and $v = v'$, respectively, where u equals t , u' equals v , and v' equals t' as strings of characters.
- (4) **congruence** `eq $t = t'$ position i follows by $\langle label \rangle$` . First call the procedure `check(Spec, t)`. Check that t and t' have the forms $\sigma(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_m)$ and $\sigma(t_1, \dots, t_{i-1}, t'_i, t_{i+1}, \dots, t_m)$, respectively. This can be done by checking the topmost operators of t and t' which must be equal in their disambiguated form, then taking the two strings between the parentheses of σ in t and t' , say α and α' , and traversing them character by character from left to right in parallel, increasing a counter whenever a left parenthesis is encountered and decreasing it whenever a right one is encountered, and also counting the number of comma characters occurring when the parenthesis counter is 0; when the number of such commas becomes $i - 1$, then extract the terms t_i and t'_i (their ends can be found by also counting parentheses), and then continue the character level equality checks for the remaining substrings as before. Once t_i and t'_i are extracted, and everything else in t and t' being checked to be identical, we check that $\langle label \rangle$ is smaller than the current label and that proof step $\langle label \rangle$ proves an equality $u = u'$, where u equals t_i and u' equals t'_i as strings of characters.
- (5) **membership** `mb $t:s$ follows by $\langle label_1 \rangle \langle label_2 \rangle$` . We check that $\langle label_1 \rangle$ and $\langle label_2 \rangle$ are smaller than the current label, and that they refer to proofs of an equational and a membership proving $u = u'$ and $v : s'$, respectively, where u equals t , u' equals v , and s' equals s as strings of characters.
- (6) **modus-ponens** `sentence axiom $\langle AxLabel \rangle$ map $v_1 : k_1 <- t_1 \dots v_V : k_V <- t_V$ follows by $\langle label_1 \rangle \dots \langle label_L \rangle$` . Identify the sentence labeled $\langle AxLabel \rangle$ in `Spec` (including the sentences added by the implication proof step), and check that its variables are exactly $v_1 : k_1, \dots, v_V : k_V$, in this order. Then for each $1 \leq i \leq V$, check that the kind of t_i is k_i and call the procedure `check(Spec, t_i)`. Then take all the atoms of the sentence labelled $\langle AxLabel \rangle$, say a, a_1, \dots, a_m , and substitute in each v_1, \dots, v_V by t_1, \dots, t_V , respectively (this is a simple word or token level substitution, because operations and symbols were always enforced to be distinct), obtaining new atoms a', a'_1, \dots, a'_m , respectively. Then check that m equals L and that each $\langle label_i \rangle$ refers to a proof step for atom a'_i for each $1 \leq i \leq L$. Finally, show that a' is identical to `sentence`, as strings of characters.

Proofs can be quite large in practical applications. Rewriting-based equational systems can perform millions of rewrites per second involving several decision procedures, such as, for example, matching routines modulo associativity, commutativity and identity, which can easily transform into tens or hundreds of millions of proof steps per second of rewriting execution. Based on experience, we claim that there are many applications that need several minutes of rewriting to be executed, so we believe that, in order to be practical and thus accepted by certifying authorities, proof checkers should not only be linear in the size of the proof objects, but should also provide potential for parallel checking. The proof checker above involves a series of small and relatively independent tasks, which makes it attractive for parallelization. However, a parallel checker needs additional efforts in order to be itself validated.

5 Synthesizing Proofs

Even though the proof objects above are human readable, we think that it is highly undesirable for humans to produce such detailed proofs manually. We next propose several techniques to automatically generate detailed proof objects as above from not so detailed proof traces produced by other systems. The crucial aspect here is how to deal with *decision procedures*, that is, how to generate detailed proofs from decision procedure computations. Since the size of proofs can easily become a bottleneck in our low level proof checking algorithm, an important issue discussed in this section is how to generate *small* proof objects.

5.1 Replacement Proofs

Perhaps the most widely known representation of an equational proof is as a *replacement proof*, that is, as a series of applications of equational axioms, either forwards or backwards, at any position in a term.

Theorem 2. $E \models (\forall X) t = t' \text{ iff } t (\Rightarrow_R \cup \Leftarrow_R)^* t'$, for any set of equations E and any terms t, t' over variables¹ in X , where R is the term rewriting system obtained from E regarding each equation as a (conditional) rewriting rule, and $\Rightarrow_R, \Leftarrow_R$ are the rewriting relation and its inverse, respectively, generated by R .

Suppose that one has a replacement proof consisting of $n - 1$ high level proof steps of the form $t_1 (\Rightarrow \cup \Leftarrow) t_2 (\Rightarrow \cup \Leftarrow) \dots (\Rightarrow \cup \Leftarrow) t_n$. Each rewriting step i , for $1 \leq i \leq n - 1$, is applied at a specific *position* p_i in t_i or in t_{i+1} , where a position is a path from the root of the term to the subterm to which an instance of the rewriting rule (or its inverse) is applied. A detailed proof object can be relatively easily generated from these higher-level replacement proofs as follows:

1. For each step $t_i \Rightarrow t_{i+1}$ involving a rewrite of l_i into r_i at position p_i in t_i of depth d_i , generate one modus-ponens step for $l_i = r_i$ followed by d_i congruences applied bottom-up along p_i , eventually proving $t_i = t_{i+1}$;
2. For each step $t_i \Leftarrow t_{i+1}$ involving a rewrite of l_i into r_i at position p_i in t_{i+1} of depth d_i , generate one modus-ponens step proving $l_i = r_i$, then one symmetry proving $r_i = l_i$ followed by d_i congruences proving $t_i = t_{i+1}$;

¹ The variables in t, t' are regarded as constants during rewriting.

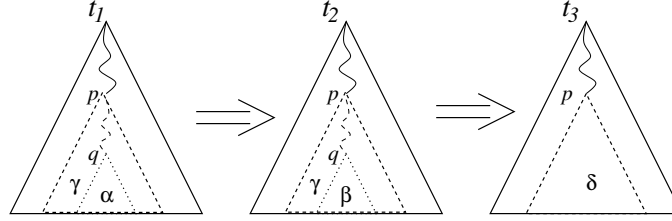
3. $n - 1$ transitivities can now generate a complete detailed proof of $t_1 = t_n$.

Therefore, for any high level replacement proof of $n - 1$ steps we can generate a detailed proof object consisting of at most $(n - 1) * (3 + \max\{\text{depth}(t_i) \mid 1 \leq i \leq n\})$ steps. The analysis above is for unconditional equations; the conditional case is more complex and we do not discuss it here.

5.2 Rewriting Proofs

Rewriting proofs are a special case of replacement proofs. More precisely, a rewriting proof of $E \models (\forall X) t = t'$ has the form $t \Rightarrow_R^* \Leftarrow_R^* t'$, where the term to which both t and t' reduce is typically in normal form. We next argue that proof objects can be generated from rewriting proofs which are typically significantly smaller than those generated from general replacement proofs.

The idea is to speculate the fact that most rewritings are applied in depth-first order, by delaying the congruence steps until all the corresponding subterms are processed. More precisely, let us consider two consecutive steps in a depth-first rewriting sequence $t_1 \Rightarrow t_2 \Rightarrow t_3$, where $t_1 = t_p[\gamma[\alpha]]$, $t_2 = t_p[\gamma[\beta]]$, and $t_3 = t_p[\delta]$, with $\alpha \rightarrow_R \beta$ and $\gamma[\beta] \rightarrow_R \delta$ two instances of rewriting rules in R :



Following blindly the procedure to generate proof objects from replacement proofs would yield the following proof steps: (1) a modus-ponens step proving $\alpha = \beta$; (2) congruence steps for all the operators on the path to position q in t_1 , in a bottom-up traversal of the path, eventually proving $t_1 = t_2$; (3) a modus-ponens step proving $\gamma[\beta] = \delta$; (4) congruence steps for the path to position p in t_2 , eventually proving $t_2 = t_3$; (5) a transitivity step proving $t_1 = t_3$. Even though this proof object is correct, a much smaller proof can be generated by delaying the application of congruence rules corresponding to positions p and above:

1. a modus-ponens step proving $\alpha = \beta$;
2. congruence steps for all the operators on the path between positions q and p in t_1 , in a bottom-up order, eventually proving $\gamma[\alpha] = \gamma[\beta]$;
3. a modus-ponens step proving $\gamma[\beta] = \delta$;
4. a transitivity step proving $\gamma[\alpha] = \delta$;
5. congruence steps for the path to position p in t_1 , eventually proving $t_1 = t_3$.

Notice that as many congruence steps as the depth of p can be saved by using this alternative procedure. Moreover, one can extend it to multiple rewriting steps in depth-first order, minimizing the application of congruence steps.

5.3 AC matching

Since the equations of associativity, commutativity and identity cannot be effectively interpreted as rewriting rules, any rewriting-based equational prover worth its salt implements efficient decision procedures for matching and rewriting modulo A, C, U and/or combinations of these. Even though rewriting engines can typically trace their applications of rewriting rules, the applications of matching decision procedures are typically opaque, thereby yielding proof gaps that need to be filled in order to generate proof objects that can be checked mechanically using external (trusted) proof checkers, such as the one described in Section 4. In this section we show how one can generate small proof objects from simplified terms which match modulo A and AC, respectively.

Let *Spec* declare a binary operator \star together with corresponding axioms of associativity and commutativity, namely “[*Assoc*] $(\forall A, B, C) (A \star B) \star C = A \star (B \star C)$ ” and “[*Comm*] $(\forall A, B) A \star B = B \star A$ ”. A \star -term can be regarded as a ground term constructed from \star together with constants (alien subterms can be handled by constant abstraction). We will assume that these constants are unique; i.e. each constant occurs at most once in any \star -term. For the purposes of proof generation this can always be arranged by suitable decoration.

Let α and β be a pair of \star -terms that are AC equivalent (this can easily be checked in practice by flattening, sorting the flattened argument lists and comparing them for equality). We next show how to construct $\mathcal{O}(n \log n)$ step proof objects of this equivalence where n is the number of constant occurrences in α (and, by AC equivalence, is also the number of constant occurrences in β).

We consider \star -terms isomorphic to (ordered) binary trees with leaves labeled by constants, and we switch between term and tree nomenclature at will. The *height* of a \star -term is the largest number of \star -occurrences above a constant.

High level proof plan. Let α_r and β_r be the right associative forms of α and β respectively. Let α_b be a balanced form of α , such that each constant in α_b occurs beneath $\lfloor \log n \rfloor$ or $\lceil \log n \rceil$ occurrences of \star . Such a balanced form can be generated algorithmically by recursive subdivision of the flattened argument list. Notice that each subterm of a balanced term is also balanced. We generate a proof that consists of four subproofs.

1. We prove $\alpha \equiv_{AC} \alpha_r$ in $\mathcal{O}(n)$ proof steps.
2. We prove $\alpha_b \equiv_{AC} \alpha_r$ in $\mathcal{O}(n)$ proof steps.
3. We prove $\alpha_b \equiv_{AC} \beta_r$ using $\mathcal{O}(n \log n)$ proof steps.
4. We prove $\beta \equiv_{AC} \beta_r$ in at most $\mathcal{O}(n)$ proof steps.

This is illustrated in Figure 1. In the case of just A matching, only steps 1. and 4. are needed because the two right parenthesized forms must be identical.

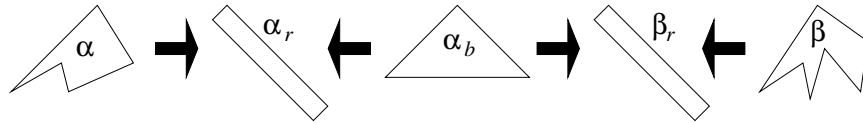


Fig. 1. Plan for proving $\alpha \equiv_{AC} \beta$.

These 4 subproofs can then be stitched together by 2 symmetry steps and 3 transitivity steps to give a proof of $\alpha = \beta$. Note that subproofs 1, 2, and 4 consist of generating the right associative form of a given term while subproof 3 requires generating the right associative form of a particular permutation, starting from a balanced form. This is the hardest subproof, and we give two distinct methods for generating it based on different sorting algorithms. We define a total ordering \prec on the constants by the left-to-right order that the constants appear in β .

Right associative form. Given an \star -term t with n constant occurrences, we consider it as a binary tree. Let m be the number of \star -occurrences on its rightmost path. Clearly $m \geq 1$. If $m = n - 1$ then t is already in right associative form. Otherwise there must exist some position p on the rightmost path such that $t|_{p,1}$ is headed by \star . We can perform a right associative proof step (a *right rotation* in the nomenclature of binary trees) at p to arrive at a new term t' whose rightmost path now contains $m + 1$ \star -occurrences, as illustrated in Figure 2.

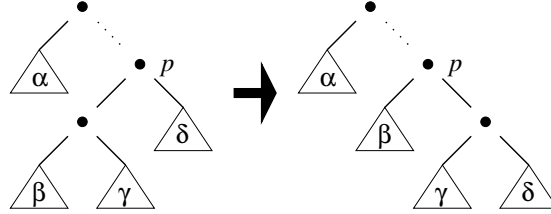


Fig. 2. Increasing the length of the rightmost path by a right rotation.

By performing this transformation $n - 1 - m$ times we must eventually arrive at right associative form and will use at most $n - 2$ associative proof steps.

Notice that each associative step needs to be followed by several congruence and transitivity steps in order to generate detailed proof objects as needed by our proof checker, and that one can generate $\Omega(n^2)$ proof steps in total if one is not careful. However, if one executes the associative steps above using an outermost strategy, then one can minimize the total number of proof steps to at most $2(n-2)$. More precisely, one can devise a procedure $\text{RIGHTASSOC}(t)$ which generates a proof for $t = t_r$, where t_r is the right associative form of t , as follows:

- if $t = (\alpha \star \beta) \star \gamma$ for some \star -terms α, β, γ , then generate a modus-ponens proof step of *Assoc* proving $(\alpha \star \beta) \star \gamma = \alpha \star (\beta \star \gamma)$, followed by a recursive call of $\text{RIGHTASSOC}(\alpha \star (\beta \star \gamma))$ which generates a proof of $\alpha \star (\beta \star \gamma) = t_r$ (in case they are not already equal), followed by a transitivity step proving $(\alpha \star \beta) \star \gamma = t_r$ (if needed);
- if $t = a \star \alpha$, where a is some constant and α is a \star -term, then recursively call $\text{RIGHTASSOC}(\alpha)$ to prove $\alpha = \alpha_r$ (in case they are different), followed by a congruence step proving $a \star \alpha = a \alpha_r$ (if needed).

Then notice that one modus-ponens and one transitivity proof steps are needed $n - 1 - m$ times, and also that at most $n - 2$ congruence steps are needed; therefore, the generated proof object will have at most $3(n - 2)$ proof steps.

Permutation via merge-sort. Let t be a \star -term in balanced form. The idea of this AC proof generator is inspired from merge sorting. Suppose some arbitrary but fixed order \prec on the constants occurring in t . Let $\text{MERGESORT}(t)$ be the procedure defined below, which, for a given balanced \star -term $t = \alpha\beta$ generates an AC proof for $t = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$, where $c_1 \prec c_2 \prec c_3 \prec \dots \prec c_n$:

1. call $\text{MERGESORT}(\alpha)$ to generate proof for $\alpha = a_1 \star (a_2 \star \dots \star a_{n_\alpha})$;
2. call $\text{MERGESORT}(\beta)$, to generate proof for $\beta = b_1 \star (b_2 \star \dots \star b_{n_\beta})$;
3. generate two congruence and one transitivity proof steps proving the equality $t = (a_1 \star (a_2 \star \dots \star a_{n_\alpha})) \star (b_1 \star (b_2 \star \dots \star b_{n_\beta}))$;
4. call the procedure $\text{MERGE}(a_1 \star (a_2 \star \dots \star a_{n_\alpha}), b_1 \star (b_2 \star \dots \star b_{n_\beta}))$ defined below, to get a proof of $(a_1 \star (a_2 \star \dots \star a_{n_\alpha})) \star (b_1 \star (b_2 \star \dots \star b_{n_\beta})) = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$;
5. generate one transitivity step proving $t = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$.

The procedure $\text{MERGE}(a_1 \star (a_2 \star \dots \star a_{n_\alpha}), b_1 \star (b_2 \star \dots \star b_{n_\beta}))$ assumes that its arguments are in right-associative sorted form, and then it generates a proof of the equality $(a_1 \star (a_2 \star \dots \star a_{n_\alpha})) \star (b_1 \star (b_2 \star \dots \star b_{n_\beta})) = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$, where the right-hand term is also in right-associative sorted form:

1. if $a_1 \prec b_1$ then call $\text{MERGE}(a_2 \star (a_3 \star \dots \star a_{n_\alpha}), b_1 \star (b_2 \star \dots \star b_{n_\beta}))$ to generate a proof of $(a_2 \star (a_3 \star \dots \star a_{n_\alpha})) \star (b_1 \star (b_2 \star \dots \star b_{n_\beta})) = c_2 \star (c_3 \star \dots \star c_n)$; then generate one modus-ponens of *Assoc*, one congruence and one transitivity proof steps that together prove that $(a_1 \star (a_2 \star \dots \star a_{n_\alpha})) \star (b_1 \star (b_2 \star \dots \star b_{n_\beta})) = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$ (notice that $a_1 = c_1$);
2. if $b_1 \prec a_1$ then call $\text{MERGE}(b_2 \star (b_3 \star \dots \star b_{n_\beta}), a_1 \star (a_2 \star \dots \star a_{n_\alpha}))$ and proceed like in 1. to generate a proof for $(b_1 \star (b_2 \star \dots \star b_{n_\beta})) \star (a_1 \star (a_2 \star \dots \star a_{n_\alpha})) = c_1 \star (c_2 \star (c_3 \star \dots \star c_n))$; one additional commutativity step followed by a transitivity prove the desired equality.

The analysis of this algorithm is straightforward: at most $5n$ proof steps are needed by MERGE , so the recurrence for MERGESORT is $T(n) = 2T(n/2) + 5n$. Therefore, the number of steps generated by MERGESORT is less than $5n \log n$.

Permutation via selection sort. We show how to reach the right associative form of an arbitrary permutation starting from a balanced form in $O(n \log n)$ proof steps. We perform $n - 1$ selection subproofs, each of which moves a chosen constant c to the next position in a growing right associative form.

These subproofs are generated by a procedure $\text{SELECT}(t)$ that takes a term $t = \alpha \star \beta$ of height $h \geq 1$ and returns a proof that $t = c \star t'$ for some t' of height $\leq h$, where c is the least constant occurring in t . We consider two cases.

1. If c occurs in α then if $\alpha = c$ our input term already has the desired form and we are done. Otherwise α has height $h - 1 \geq 1$ and we call $\text{SELECT}(\alpha)$ to get a proof that $\alpha = c \star \alpha'$ for some α' of height $\leq h - 1$. We use a congruence step to show that $\alpha \star \beta = (c \star \alpha') \star \beta$. Substituting in the associative axiom we prove $(c \star \alpha') \star \beta = c \star (\alpha' \star \beta)$. Finally with a transitivity step we prove $\alpha \star \beta = c \star (\alpha' \star \beta)$. This is illustrated in Figure 3.
2. If c occurs in β then we use prove $\alpha \star \beta = \beta \star \alpha$ by substituting in the commutative axiom. We then follow the method of case 1 with the rôles of α and β reversed to prove that $\beta \star \alpha = c \star (\beta' \star \alpha)$ for some β' of height $h - 1$. Finally with a transitivity step we prove $\alpha \star \beta = c \star (\beta' \star \alpha)$.

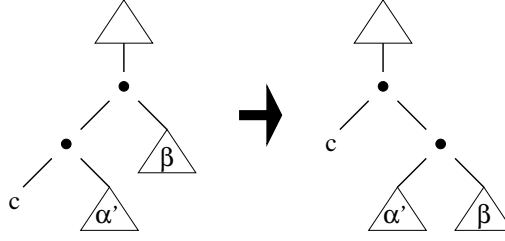


Fig. 3. Case 1 for SELECT

Each subproof requires at most $5h$ steps however assembling to subproofs into the final proof must be done with some care to avoid a quadratic number of congruence steps. The key idea is to generate the subproofs top-down but stitch them together with congruence and transitivity steps bottom-up. This is done by a procedure $\text{SORT}(t)$ which takes a term t of height h and returns its sorted right associative form. We consider two cases.

1. If t is a lone constant then we are done.
2. Otherwise we use $\text{SELECT}(t)$ to get a proof that $t = c \star t'$ where c is the least constant in t . We then use $\text{SORT}(t')$ to get a proof that $t' = t''$ where t'' is the sorted right associative form of t' . Using a congruence step we prove that $c \star t' = c \star t''$ and finally a transitivity step proves $t = c \star t''$.

SORT makes $n - 1$ calls to SELECT and performs $n - 1$ congruence steps and $n - 1$ transitivity steps. Since $h \leq \lceil \log n \rceil$ we have an upper bound on the number of proof steps required of $5(n - 1)\lceil \log n \rceil + 2(n - 1)$.

5.4 Arbitrary Combinations of A, C and U Matching

We next assume that procedures generating proofs from A and AC matchings for ground terms constructed from a single binary operator together with unique constants are available (efficient such procedures were presented in the previous subsection), and informally describe a method by which these procedures can be combined to generate proof objects of size $\mathcal{O}(n \log n)$ for terms that match due to any combination of A, C and/or U at any positions and for any operators.

Suppose that t and t' are two ground terms involving several operators, each being potentially associative, commutative and/or have identity. We first associate to t and t' datastructures $d(t)$ and $d(t')$, respectively, which replace multiple consecutive occurrences of the same associative or associative and commutative operator by a list tagged with the operator name and containing all the corresponding subterms. Second, we eliminate all the constants (together with their immediately above binary operator) which are identities for operators, but only if they occur immediately under the operations for which they are identities; we need to store information regarding the places from which these identities have been removed, because appropriate identity proof steps have to be generated later. Finally, we *sort the subterms* of each commutative operator or list corresponding to an associative and commutative operator, storing information regarding the permutation of subterms that lead to the sorted order; notice that sorting must be performed in a bottom-up fashion on the datastructures, and

use some conventions (arbitrary but fixed; for example, one can use lexicographic order) on how to compare lists of terms. $d(t)$ and $d(t')$ should be identical now; otherwise an error stating that t and t' are not equal modulo combinations of A, C and/or U axioms should be generated. By composing the permutations of each sorted list at each appropriate position in $d(t)$, one obtains a one-to-one map between subterms in t and subterms in t' ; equality proof obligations will be generated for subterms in this relation.

We can now start generating the proof object. Traverse $d(t)$ bottom-up and for each node do the following: (1) if the node is a list corresponding to an associative or an associative and commutative operator then, then by calling the appropriate A or AC matching procedure on the subterms of t and t' from which the sorted list node in $d(t)$ has been formed, where all the alien subterms are replaced by some artificial distinct constants, we generate a proof of equality for those subterms of t and t' ; (2) if the node is a commutative operator then we generate a commutativity proof step only if needed, i.e., if the two subterms occur in different order in t and t' . As the datastructure is traversed bottom-up, appropriate congruence proof steps need to be generated, as well as identity proofs for those nodes where an identity constant has been removed. Notice that only a linear number of proof steps is added to the ones already generated by the A and AC procedures. Therefore, the size of the proof is still $\mathcal{O}(n \log n)$ in the size n of t and t' .

5.5 System Dependent Aspects

The proof synthesis issues discussed so far are general to all equational provers. In this subsection we discuss some aspects which are more system dependent; our case study is Maude [5].

Disambiguation and desugaring. As seen in Section 3, disambiguated specifications, sentences and proofs look ugly and unreadable. As a convenience to users, rewriting engines and theorem provers typically use a “sugared” and sometimes apparently ambiguous syntax. However, in order for the proof checker presented in Section 4 to be as simple as possible, all the symbols and statements are required to be completely disambiguated and desugared. We next discuss how this can be done in the case of Maude.

Kinds are not declared explicitly in Maude. Users only declare sorts and relate them via subsorts. Kinds are then generated automatically from the sort partial order, one kind per connected component. For s a given sort, $[s]$ refers to the kind to which s belongs. Therefore, a Maude desugarer needs to generate one kind per connected component and replace each occurrence of $[s]$ by the appropriate kind. Moreover, each subsort relation “ $s < s'$ ” in kind k needs to be replaced by a conditional membership “ $(\forall x : k) x : s' \text{ if } x : s$ ”, and each operation declaration needs to be replaced by a disambiguated kind level operation declaration and an appropriate conditional membership (if needed). For example, $\sigma : s_1 \times \dots \times s_n \rightarrow s$ needs to be replaced by $\sigma_{k_1 \dots k_n, k} : k_1 \times \dots \times k_n \rightarrow k$ and $(\forall x_1 : k_1, \dots, x_n : k_n) \sigma_{k_1 \dots k_n, k}(x_1, \dots, x_n) : s \text{ if } x_1 : s_1, \dots, x_n : s_n$. Notice that some, or even all, of the arguments of an operation can be already

declared as kinds (using the bracket notation). Only those arguments which are declared as sorts need a variable and an appropriate condition as above. A sentence is not needed for an operation only if all the arguments and the result of that operation are declared as kinds.

Another easy to desugar feature in Maude is the declaration of variables in sentences; these can also be declared as having sorts rather than kinds in Maude. Each sentence using variables in sort notation needs to be desugared by declaring those as kinded variables and by adding appropriate conditions to the sentence. For example, the sentence $(\forall x_1 : s_1, \dots, x_n : s_n) a \text{ if } a_1, \dots, a_n$ is replaced by the sentence $(\forall x_1 : k_1, \dots, x_n : k_n) a \text{ if } a_1, \dots, a_n, x_1 : s_1, \dots, x_n : s_n$.

The harder part is to disambiguate operations occurring in terms in equations, memberships, or in high-level proofs given as Maude traces. A simple solution would, of course, be to implement the disambiguater as part of Maude and then provide an auxiliary Maude command or flag to disambiguate all operators. Even though this could probably be done, there is an alternative solution which would only require a small change in Maude, namely to tag each operation by its *result kind* only. Since variables also have a unique kind, one can easily devise an algorithm now that disambiguates any term.

Sharing and memoization. A major reason for which we designed a low level proof representation and checker in Sections 3 and 4, besides generality and simplicity, was also to deal properly with sharing and memoization. An important optimization of rewriting engines, also supported by Maude, is to avoid redoing computations. A typical example occurs when the right-hand term of a rewriting rule contains multiple copies of the same subterm. Then, in any given instance of that rule, the shared subterm is reduced only once to its normal form, which is then copied multiple times in the resulting term. Another typical example involves memoization, or caching. In Maude, some operations can be declared with the attribute `memo`, meaning that the normal forms of terms having those operations at top are cached and never reduced again.

Even though sharing and/or memoization could probably be replaced in some complicated way by actual proof steps in a replacement or rewriting equational proof, we believe that such attempts would be not only artificial and error-prone, but also quite impractical because they would increase the size of proofs significantly. Our current formalization of proofs in Section 3 supports sharing and memoization naturally, because one can just refer to the appropriate (previous) proof step label when a shared sentence has been already derived. Sharing does not explicitly occurs in Maude's current execution traces; its existence can be inferred by just noticing gaps in traces. However, one can easily implement an independent algorithm which "fills the gaps" in Maude's traces and thus generate detailed proof objects as needed by our certifier: whenever a gap is noticed in a Maude trace (which is not given to an AC match - gap that can be filled using the algorithms in Subsection 5.3), search through the already generated proof object for a fitting sentence (such a sentence must exist, because otherwise Maude could not have generated the trace) and then refer to it by its label when generating the corresponding congruence rules.

Least sort calculation. Due to overloading operators, the same term can have multiple result sorts. For example, $3 + 5$ can be a natural, an integer, or a rational number. Based on the partial order on sorts, Maude implements efficient decision procedures to calculate the least sort of any term. A term is substituted for a variable in a rewriting rule application only if the sort of the term is smaller or equal to the sort of the variable. These implicit applications of conditional membership rules need to be explicitly stated when generating the proof object. Again, this can be relatively easily done, without modifying Maude in any essential way, if Maude is slightly changed to report a resulting *sort* for each operation use in each term; this is a slightly stronger but more general requirement than the one needed for disambiguation and desugaring (where only the kind result of operations was needed).

6 Conclusion and Future Work

We have presented a general method to certify and synthesize proofs in membership equational logic, where the synthesis may involve generating full proofs from proof traces modulo combinations of A, C, and U axioms. We have proposed a simple representation of proof objects and have given algorithms that can synthesize short proofs for the \mathcal{AX} -gaps. Our proof representations and algorithms are quite close to an actual implementation of both a generic proof checker and of a synthesis tool to generate proof objects from Maude traces. We plan to develop both tools in the near future. This will provide certified proof objects for all equational computations in Maude. Since membership equational logic is a sublogic of rewriting logic [4], our work has a very natural extension to certification and synthesis of rewriting logic proofs. Therefore, a subsequent development will involve investigating such an extension. Since rewriting logic has good properties as a logical framework [10], the extension to rewriting logic is potentially interesting not only for interoperating Maude with other tools, but also to represent proof objects from different logics.

Acknowledgements Grigore Roşu was supported in part by joint NSF/NASA grant CCR-0234524. Steven Eker was supported DARPA through Air Force Research Laboratory Contract F30602-02-C-0130. Patrick Lincoln was supported by ONR Grants N00014-01-1-0837, N00014-02-1-0109, and N00014-01-1-0795. José Meseguer was supported by ONR Grant N00014-02-1-0715.

References

1. H. Barendregt and E. Barendsen. Autarkik computations and formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.
2. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
3. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
4. R. Bruni and J. Meseguer. Generalized rewrite theories. Manuscript, January 2003, <http://maude.cs.uiuc.edu>.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1987.
7. J. Goguen and G. Roşu. Institution morphisms. *Formal Aspects of Computing*, 13(3–5):274–307, 2002.
8. M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
9. M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.
10. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.
11. J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium’87*, pages 275–329. North-Holland, 1989.
12. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
13. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT’97*, pages 18–61. Springer LNCS 1376, 1998.
14. J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, May/June 1994*, pages 48–80. Springer LNCS 906, 1995.
15. T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In *Proceedings of FOSSACS’02: Foundations of Software Science and Computational Structures*, pages 326–341. Springer LNCS 2303, 2002.
16. Q. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3–4):309–336, 2002.
17. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
18. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV ’96*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
19. G. Roşu. Complete categorical equational deduction. In L. Fribourg, editor, *Proceedings of Computer Science Logic (CSL’01)*, volume 2142 of *Lecture Notes in Computer Science*, pages 528–538. Springer, 2001.
20. M.-O. Stehr. Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory. Doctoral Thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. <http://www.sub.uni-hamburg.de/disse/810/>.
21. M.-O. Stehr, P. Naumov, and J. Meseguer. The HOL/NuPRL proof translator—A practical approach to formal interoperability. In *Proc. 14th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOL’2001) Edinburgh, Scotland, September 2001*, pages 329–345. Springer LNCS 2152, 2001.
22. A. Tarlecki. Towards heterogeneous specifications. In *Proc. Workshop on Frontiers of Combining Systems FroCoS’98, Amsterdam, October 1998*. Applied Logic Series, Kluwer Academic Publishers, 1998.