A Two-Level Approach towards Lean Proof-Checking

Gilles Barthe*, Mark Ruys and Henk Barendregt

May 15, 1996

Abstract

We present a simple and effective methodology for equational reasoning in proof checkers. The method is based on a two-level approach distinguishing between syntax and semantics of mathematical theories. The method is very general and can be carried out in any type system with inductive and oracle types. The potential of our two-level approach is illustrated by some examples developed in Lego.

1 Introduction

The main actions in writing mathematics consist of defining, reasoning and computing (symbolically; this is also called 'equational reasoning'). Whereas defining and reasoning are reasonably well captured by an interactive proof-developer, the formalization of computations has caused problems. This paper studies the possibilities of a partial automation of equational reasoning, which is from the authors' experience, one of the most recurrent source of problems in formalizing mathematics using a proof-developer [5, 25]. We describe several methods using elementary techniques from universal algebra which provides an efficient tool to solve problems of an equational nature in any type theory with inductive types and term-rewriting (inductive types are required for a formalization of universal algebra, in particular for the formalization of the type of terms of a signature).

Our main goal is to solve equational problems of the form $a =_{\mathcal{A}} b$, where $\mathcal{A}$ is a model of a given equational theory $\mathcal{S} = (\Sigma, E)$, a and b are (expressions for) elements of $\mathcal{A}$, and $=_{\mathcal{A}}$ is the equality relation of the carrier of $\mathcal{A}$. To do so, we use two naming principles:

for satisfiability: we recast the problem $a =_{\mathcal{A}} b$ in a syntactic form $[\![\ulcorner a \urcorner]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![\ulcorner b \urcorner]\!]^{\mathcal{A}}_{\alpha}$ where α is an assignment and $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ are two Σ-terms such that

$$[\![\ulcorner a \urcorner]\!]^{\mathcal{A}}_{\alpha} = a \quad \text{and} \quad [\![\ulcorner b \urcorner]\!]^{\mathcal{A}}_{\alpha} = b$$

*Current address: Department of Software Technology, CWI, Amsterdam, The Netherlands.

where $[\![\_]\!]_\alpha^\mathcal{A}$ denotes the α-interpretation of Σ-terms into the model $\mathcal{A}$. (Note that such terms always exist and one can even find optimal terms). By the soundness theorem, the latter problem follows from $\mathcal{S} \vdash \ulcorner a \urcorner \doteq \ulcorner b \urcorner$ (we use this informal notation to state that $(\ulcorner a \urcorner, \ulcorner b \urcorner)$ is a theorem of $\mathcal{S}$). If $\mathcal{S}$ is equivalent to a canonical term-rewriting system $\mathcal{R}$, then the last problem can be solved automatically by taking the $\mathcal{R}$-normal forms of $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ and check whether they are equal. We internalize the whole informal process using *oracle types* [7]; the rewrite system is grafted to the type theory in such a way that the conversion rule itself is changed and checking whether $[\ulcorner a \urcorner] = [\ulcorner b \urcorner]$ (the equality here is Leibniz equality) boils down to a reflexivity test, which can be done by the proof checker.

for extensionality: often we need a proof object for statements of the form

$$s =_\mathcal{A} t \quad \Rightarrow \quad \phi(s) =_\mathcal{A} \phi(t) \tag{1}$$

where s, t and $\phi(x)$ be (expressions for) elements of $\mathcal{A}$. If this is done in the way taught in books on logic (applying several times the axioms of equational logic) a proof object for this fact becomes rather large: quadratic in the size of the expression 'ϕ'. However, using the naming principle one can solve (1) by proving the meta-result

$$s =_\mathcal{A} t \quad \Rightarrow \quad [\![\ulcorner \phi \urcorner]\!]^\mathcal{A}_{\alpha\,(x:=s)} =_\mathcal{A} [\![\ulcorner \phi \urcorner]\!]^\mathcal{A}_{\alpha\,(x:=t)}$$

for all $\ulcorner \phi \urcorner$. This result has a proof of fixed size.

In this paper, we shall give a detailed presentation of these methods (and some minor variants) and demonstrate with non-trivial examples that they provide a suitable tool for a partial automation of equational reasoning in proof-checking. The distinctive features of our approach are:

- it applies to type systems where equality is treated axiomatically (intensional frameworks) and with proof-objects; the only requirement is the presence of (first-order) inductive types and so-called oracle types;
- the size of the implementation of the proof-checker is kept fairly small; the whole process can be carried out within the proof-checker;
- the proof-checker is built upon formal systems whose meta-theory is easy to understand.

The paper is organized as follows: in section 2, we introduce the relevant mathematical background for the subsequent parts of the paper. In section 3, we specify the nature of equational reasoning and delimit the range of equational problems whose resolution can be automated. In section 4, we discuss the possible approaches to the automation of equational reasoning and present our own solution in terms of oracle types. In section 5, we present a preliminary implementation of the two-level approach in Lego. Large parts of the paper are of expository nature; they have been included because (i) the material we present has never been presented elsewhere with a view to use it for our specific purpose (ii) the main contribution of this paper is to specify the problem and device a methodology to solve it (but the methodology does not use any new technique).

The two-level approach was grew out from earlier work by P. Aczel and the first author on the formalization of (universal) algebra in type theory. The applications of universal algebra for equational reasoning were realized later by the first author and presented at the HISC meeting in Amsterdam in March 1994 (see [5, 4]). After the completion of this work, H. Elbers and the first author have developed further the two-level approach and provided an automatic procedure to solve equational problems in Lego [6]. The work presented in this paper bears some similarities with the work of the NuPrl team on reflection [11, 16], although the specific use of naming principles to automate equational reasoning seems to be new.

Acknowledgments Thanks to P. Aczel, H. Elbers and H. Geuvers for useful discussions on the two-level approach. Thanks to J. Harrison for his comments on an earlier version of the draft. This work was partially supported by the ESPRIT project 'TYPES: Types for Programs and Proofs'.

2 Mathematical Background

In this section, we review some standard material on equational logic and term-rewriting. During the last few years, there has been an explosion in the number of variants of equational logic: many-sorted, order-sorted, conditional... We shall only be concerned with the simplest formalism, unsorted equational logic. For convenience, we separate the presentation in two parts; the first part is concerned with syntax, equational deduction and term-rewriting. The second part is devoted to semantics. See [10, 18] for a longer introduction to the notions involved.

2.1 Equational Logic and Term-Rewriting

The basic notions of universal algebra are those of signature and equational theory. As the notions are standard, we give them without any further comment.

Definition 1 – *A* signature *is a pair* $\Sigma = (F_\Sigma, \mathsf{Ar})$ *where* F_Σ *is a set of function symbols and* $\mathsf{Ar} : F_\Sigma \to \mathbb{N}$ *is the arity map.*

- *Let* Σ *be a signature. Let* V *be a fixed, countably infinite set of variables. The set* T_Σ *of* Σ-terms *is defined as follows:*
  - *if* $x \in V$*, then* $x \in T_\Sigma$*,*
  - *if* $f \in F_\Sigma$ *and* $t_1, \ldots, t_{\mathsf{Ar} f} \in T_\Sigma$*, then* $f(t_1, \ldots, t_{\mathsf{Ar} f}) \in T_\Sigma$*.*
- *A map* $\theta : T_\Sigma \to T_\Sigma$ *is a* Σ-substitution *if for every* $f \in F_\Sigma$ *and* Σ*-terms* $t_1, \ldots, t_{\mathsf{Ar} f}$ *we have* $\theta(f(t_1, \ldots, t_{\mathsf{Ar} f})) = f(\theta t_1, \ldots, \theta t_{\mathsf{Ar} f})$*.*

- *The relation $\leq$ is defined by $t, t' \in T_\Sigma$, $t \leq t'$ if there exists θ such that $\theta t = t'$. The pre-order induced by $\leq$ is denoted by $T_\Sigma^{\leq}$.*
- *The set* $\mathsf{var}(s)$ *of variables of a term s is defined inductively as follows:*
  - *if $x \in V$, then* $\mathsf{var}(x) = \{x\}$,
  - $\mathsf{var}(f(t_1, \ldots, t_{\mathsf{Ar}f})) = \bigcup_{1 \leq i \leq n} \mathsf{var}(t_i)$.
- *if s and t are Σ-terms and u is an occurrence of s, $s[u \leftarrow t]$ is the term obtained by replacing the subterm of s at u by t.*

Note that every (partial) map $\theta : V \rightarrow T_\Sigma$ yields a Σ-*substitution* in an obvious way. We shall sometimes refer to such maps as partial substitutions. The standard terminology can be carried over to partial substitutions, so we will also talk about partial renamings.

Equational Logic. A Σ-*equation* is a pair of Σ-terms (s, t), usually written as $s \doteq t$.

Definition 2 *An* equational theory *is a pair $\mathcal{S} = (\Sigma, E)$ where Σ is a signature and E is a set of Σ-equations.*

The rules for equational deduction are given in the following table:

Rules for equational deduction	
$\dfrac{}{s \doteq s}$	Reflexivity
$\dfrac{s \doteq t}{t \doteq s}$	Symmetry
$\dfrac{s \doteq t \quad t \doteq u}{s \doteq u}$	Transitivity
$\dfrac{s_1 \doteq t_1 \ldots s_n \doteq t_n}{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots t_n)}$	Compatibility
$\dfrac{s \doteq t}{\theta s \doteq \theta t}$	Instantiation

where θ is a substitution.

Definition 3 *Let $\mathcal{S} = (\Sigma, E)$ be an equational theory. A Σ-equation $s \doteq t$ is a* theorem *of $\mathcal{S}$ (written $\mathcal{S} \vdash s \doteq t$) if it is deducible from E using the rules for equational deduction.*

Term-Rewriting. Let Σ be a signature.

Definition 4 *A Σ-*rewrite rule *is a pair of Σ-terms (s, t), usually written $s \rightarrow t$, such that s is a non-variable term and* $\mathsf{var}(t) \subseteq \mathsf{var}(s)$. *A Σ-*rewrite system *is a set of rewrite rules.*

As usual, we talk about rewrite rules and rewrite systems when there is no risk of confusion. Note that every Σ-rewrite system $\mathcal{R}$ induces an equational theory $(\Sigma, \mathcal{R})$, simply by seeing rewrite rules as equations. By *abus de notation*, we shall denote this equational theory by $\mathcal{R}$.

Let $\mathcal{R}$ be a rewrite system and s and t be two Σ-terms. We say that s *one step $\mathcal{R}$-rewrites* to t (notation $s \to_{\mathcal{R}} t$) if there exist an occurrence u of s, a rewrite rule (l, r) in $\mathcal{R}$ and a Σ-substitution θ satisfying $s/u = \theta l$ and $t = s[u \leftarrow \theta r]$.

We let $\twoheadrightarrow_{\mathcal{R}}$ and $\leftrightarrow_{\mathcal{R}}$ be respectively the reflexive transitive and the reflexive, symmetric and transitive closure of $\to_{\mathcal{R}}$. Finally, $s \downarrow_{\mathcal{R}} t$ if there exists u such that $s \twoheadrightarrow_{\mathcal{R}} u$ and $t \twoheadrightarrow_{\mathcal{R}} u$. Note that $\downarrow_{\mathcal{R}} \subseteq \leftrightarrow_{\mathcal{R}}$.

Definition 5 *A rewrite system $\mathcal{R}$ is* confluent *if $\downarrow_{\mathcal{R}} = \leftrightarrow_{\mathcal{R}}$ and* terminating *if there is no infinite reduction sequence $t \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$. A rewrite system is* canonical *if it is both confluent and terminating.*

Proposition 6 *Let $\mathcal{R}$ be a confluent rewrite system.*

$$(s \downarrow_{\mathcal{R}} t) \quad \Leftrightarrow \quad (s \leftrightarrow_{\mathcal{R}} t) \quad \Leftrightarrow \quad \mathcal{R} \vdash s \doteq t \ .$$

Remark. Algebraic structures are usually described equationally rather than as term-rewriting systems. However, some of them can be turned into term-rewriting systems using the Knuth-Bendix completion procedures [18].

2.2 The Semantics of Equational Logic and the Completeness Theorem

Equational theories are syntactical descriptions of mathematical objects. The objects satisfying these descriptions are the mathematical structures themselves. In this section, we define a semantics for equational theories. As we are interested in using universal algebra to solve the problem of equational reasoning in type theory, our semantics is ultra-loose, i.e. the equality relation between terms is interpreted as an arbitrary equivalence relation rather than as the underlying equality of the model.

Definition 7 *An* Σ-algebra $\mathcal{A}$ *for a signature Σ consists of a set A, an equivalence relation $=_{\mathcal{A}}$ on A and for each function symbol f of arity n, a function $f^{\mathcal{A}} : A^n \to A$ such that for every $(a_1, \ldots, a_n), (a'_1, \ldots, a'_n) \in A^n$,*

$$a_1 =_{\mathcal{A}} a'_1, \ldots, a_n =_{\mathcal{A}} a'_n \quad \Rightarrow \quad f^{\mathcal{A}}(a_1, \ldots, a_n) =_{\mathcal{A}} f^{\mathcal{A}}(a'_1, \ldots, a'_n) \ .$$

For implementation purposes, we us a slightly modified definition of assignment and satisfiability. Of course, the resulting semantics is equivalent to the standard one.

Definition 8 *An $\mathcal{A}$-*assignment *is a partial map $\alpha : V \rightharpoonup A$ with a non-empty, finite domain.*

Any $\mathcal{A}$-assignment can be extended inductively to a partial function $[\![\_]\!]_\alpha^\mathcal{A}$ on the set of Σ-terms:

$$\begin{array}{rcll} [\![x]\!]_\alpha^\mathcal{A} & \simeq & \alpha x & \text{if } x \in \mathsf{dom}\ \alpha \\ [\![f(t_1,\ldots,t_n)]\!]_\alpha^\mathcal{A} & \simeq & f^\mathcal{A}([\![t_1]\!]_\alpha^\mathcal{A},\ldots,[\![t_n]\!]_\alpha^\mathcal{A}) & . \end{array}$$

Definition 9 *Let $\mathcal{A}$ be a Σ-algebra. Two $\mathcal{A}$-assignments α and β are* compatible *if* $\mathsf{dom}\ \alpha = \mathsf{dom}\ \beta$ *and $\alpha x =_\mathcal{A} \beta x$ for all $x \in \mathsf{dom}\ \alpha$.*

The following lemma shows that compatible assignments satisfy the same equations.

Lemma 10 (Compatibility lemma) *Let $\mathcal{A}$ be a Σ-algebra. Let α and β be two compatible $\mathcal{A}$-assignments. Let t be a Σ-term such that* $\mathsf{var}(t) \subseteq \mathsf{dom}\ \alpha$. *Then $[\![t]\!]_\alpha^\mathcal{A} =_\mathcal{A} [\![t]\!]_\beta^\mathcal{A}$.*

We write $\mathcal{A} \models s \doteq t$ if for all $\mathcal{A}$-assignments α such that $\mathsf{var}(s) \cup \mathsf{var}(t) \subseteq \mathsf{dom}\ \alpha$,

$$[\![s]\!]_\alpha^\mathcal{A} =_\mathcal{A} [\![t]\!]_\alpha^\mathcal{A} \ .$$

Definition 11 *Let $\mathcal{S} = (\Sigma, E)$ be an equational theory. A Σ-algebra $\mathcal{A}$ is a $\mathcal{S}$-model if $\mathcal{A} \models s \doteq t$ for all the equations $s \doteq t$ in E.*

We say that $\mathcal{S} = (\Sigma, E)$ *semantically entails* a Σ-equation $s \doteq t$ (notation $\mathcal{S} \models s \doteq t$) if $\mathcal{A} \models s \doteq t$ for every $\mathcal{S}$-model $\mathcal{A}$. The fundamental theorem of equational logic establishes the compatibility between syntax and semantics.

Theorem 12 (Soundness/Completeness) *For every Σ-equation $s \doteq t$,*

$$\mathcal{S} \vdash s \doteq t \quad \Leftrightarrow \quad \mathcal{S} \models s \doteq t \ .$$

The completeness result is proved by constructing the term-model $T_\mathcal{S}$ as the quotient of T_Σ by the provability relation $\sim_\mathcal{S}$. The crucial fact that we shall exploit later is that for every term s and t,

$$\mathcal{S} \vdash s \doteq t \quad \Leftrightarrow \quad [s] = [t]$$

where $[\_] : T_\Sigma \to T_\mathcal{S}$ is the canonical map assigning to every term its equivalence class under the provability relation.

3 The Naming Principles

In this section, we define a methodology to solve equational problems in type theory. Our methodology is very flexible and can be carried out in any type system with inductive types. In particular, it can be carried out in the underlying type systems of Lego [20], Coq [13], Alf [21] and NuPrl [12].

3.1 Specifying the Problem to be Solved

Our first task is to fix the boundaries of the problem to be solved. In its most general form, equational reasoning is concerned with determining whether two elements s and t of a set V of values are related by an equality relation R. Naturally, the problem is far too general to have an automated solution. Yet there is a well-understood branch of mathematical logic, namely equational logic, which is concerned with equational theories, i.e. first-order languages with a single (binary) predicate symbol $=$. Equational logic provides the *right level of generality* to tackle the problem of equational reasoning for several reasons:

1. the problem is general enough: a wide collection of mathematical theories can be presented equationally, for example the theories of monoids, groups and rings;
2. one might expect an useful and automated solution to the problem: in some cases, it is possible to provide an algorithm to test whether an equation of a given theory $\mathcal{S}$ is a theorem of this theory;
3. this work can provide a theoretical foundation to integrate computer algebra systems and proof checkers: computer algebra systems, with their impressive power, are mostly concerned with equational theories.

This justifies the following choice for the form of an equational problem.

The problem. *Let $\mathcal{S}$ be an equational theory. Let $\mathcal{A}$ be a model of $\mathcal{S}$. Let a and b be expressions for elements of $\mathcal{A}$. Does $a =_{\mathcal{A}} b$?*

Note that the problem makes sense within a type system with inductive types as one can formalize all basic notions of universal algebra in such a system. Here are a few examples of equational problems.

Example 13
- *Let $\mathbb{Z}_n$ be the ring of integers modulo n, where $n \geq 3$. Does $2(n-1) = 0$?*
- *Let $\boldsymbol{D}_8$ be the dihedral group with eight elements. Let $\sigma, \tau \in \boldsymbol{D}_8$. Does $\tau\sigma = \sigma^3\tau$? Here the problem is quantified over all elements of $\boldsymbol{D}_8$.*
- *Let $(M, =_M, \circ_M, e_M)$ be a monoid. Let $x, y \in M$. Does $(x \circ_M e) \circ_M y =_M x \circ_M y$? Here the problem is quantified over all $x, y \in M$ and monoids M.*

To solve the problem, we will first relate it to equational logic and then use equational logic to solve the problem automatically.

For the remaining of this section, we work with the formalization of universal algebra in the type system. In particular, an equational theory is an inhabitant of the type of equational theories, and a model of a theory is an inhabitant of the type of models of this theory. To alleviate the presentation, we will still use the ordinary language of universal algebra.

In the sequel, we let $\mathcal{S} = (\Sigma, E)$ be a fixed equational theory and $\mathcal{A}$ be a model of $\mathcal{S}$.

3.2 Equational Logic, Local Equational Logic and Equational Reasoning

Equational logic is *global* in the sense that it is used to determine whether a $\mathcal{S}$-equation $s \doteq t$ is true in all models of $\mathcal{S}$, i.e. whether $\mathcal{S} \models s \doteq t$. In contrast, equational reasoning is *local*, in the sense that one is also interested whether a given equality holds in a specific model, i.e. $a =_{\mathcal{A}} b$ for some specific a and b in a specific model $\mathcal{A}$ of $\mathcal{S}$. An intermediate formal system is *local equational logic*, a variant of equational logic whose deductive system allows to infer whether $\mathcal{A} \models s \doteq t$ for a specific model $\mathcal{A}$ of $\mathcal{S}$. One could even go one step further and develop a formal system to infer whether $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$ in a specific model $\mathcal{A}$ and for a specific assignment α. This last problem, which we call the *local satisfiability problem* is in fact a special instance of equational reasoning. If we analyse the logical formulations of local satisfiability and semantical entailment, we see that the latter represents an uniform notion of the former[1]. One concludes that *the goal of equational logic is to know whether an uniform collection of equational problems is satisfied.*

Local satisfiability is a very common form of equational problem. However, not all equational problems arising in the formalization of mathematics are concerned with local satisfiability. An equally important instance of equational problem is the *extensionality problem*: given a $\mathcal{S}$-term t, a model $\mathcal{A}$ of $\mathcal{S}$ and two interpretations α, β in $\mathcal{A}$, does $[\![t]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\beta}$? In fact, those two problems (local satisfiability and extensionality) form the core of equational reasoning.

3.3 The Naming Principles

As outlined in the previous subsection, there is a divergence between equational logic as a formal system and equational reasoning as it occurs in mathematics. We have

a goal: an equational problem, i.e. an equality $a =_{\mathcal{A}} b$;

some tools: equational logic, which can be used to solve a local satisfiability problem, and the compatibility lemma, which can be used to solve an extensionality problem.

The difficulty in applying the tools to solve the goal is that equational problems are essentially of a semantical nature while equational logic is designed to solve syntactical problems. In order to apply equational logic to equational reasoning, one must perform a preliminary manipulation on equational problems, so that they present themselves in a form which is amenable to be solved by equational logic. What is needed here is a *naming principle* which transforms a semantical equational problem into a local satisfiability problem or an extensionality problem. For the clarity of the discussion, we will therefore distinguish

[1] By the soundness/completeness theorem, $\mathcal{S} \models s \doteq t$ is equivalent to the collection of local satisfiability problems $([\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha})_{(\mathcal{A} \in \mathcal{M}, \alpha \in \mathcal{V}(\mathcal{A})}$ where $\mathcal{M}$ is the collection of $\mathcal{S}$-models and for $\mathcal{A} \in \mathcal{M}$, $\mathcal{V}(\mathcal{A})$ is the set of $\mathcal{A}$-assignments.

between the *naming principle for satisfiability* (for short NPS) and the *naming principle for extensionality* (for short NPE). One fundamental feature of these naming principles is that they do not require any extension of the type system; indeed, the naming principles are a special instance of conversion rules. We introduce these principles below.

3.3.1 The Naming Principle for Satisfiability.

The aim of the naming principle for satisfiability is to recast a local equation $a =_{\mathcal{A}} b$ into an equation of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$, where

- s and t are terms of the theory T,
- $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,
- $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow b$.

Of course, the equation to be solved has not changed; what has changed is the way to look at it. The equation in its second form makes it clear that the problem to be solved is an instance of an uniform collection of equational problems, as defined in the previous section. The advantage of this switch of perspective is that the equation in its second form is more amenable to be solved by standard syntactic tools. Indeed, $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$ is an immediate consequence of $\mathcal{S} \vdash s \doteq t$. This yields a semi-complete[2] method to prove $a =_{\mathcal{A}} b$:

1. apply the NPS; this reduces the equational problem to one of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$;
2. apply any method available to prove $\mathcal{S} \vdash s \doteq t$.

Of course, the efficiency of the method depends on the choice of s and t[3]. Fortunately, there is always an optimal application of the NPS.

Definition 14 *Let $\mathcal{A}$ be a model of $\mathcal{S}$. Let a be an element of $\mathcal{A}$. The* pre-order of codes *of a is the sub-pre-order of $T^{\leq}_{\Sigma}$ whose elements are the terms t for which there exists an assignment α such that $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$.*

For every element a of $\mathcal{A}$, the pre-order of codes of a has a top element (unique up to renaming), called the *optimal code* of a. We write $\ulcorner a \urcorner$ for the optimal code of a.

Similarly, we can define a *code for an equational problem $a =_{\mathcal{A}} b$* to be an equation $s \doteq t$ such that for some assignment α, $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$ and $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$. Every equational problem $a =_{\mathcal{A}} b$ has an *optimal code* $\ulcorner a \urcorner \doteq \ulcorner b \urcorner$ (one can verify that $\ulcorner a \urcorner$ and $\ulcorner b \urcorner$ are optimal codes for a and b respectively) with the two properties:

[2] The method can fail even if the equational problem is true.

[3] Indeed, some uses of the NPS can be less than judicious. Every equational problem $a =_{\mathcal{A}} b$ can be reduced by the NPS to $[\![s]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\alpha}$ where s and t are distinct variables and α is any assignment satisfying $\alpha s = a$ and $\alpha t = b$. In order to solve the problem according to the proposed method, we must now solve $\mathcal{S} \vdash s \doteq t$. This only holds if the theory is inconsistent!

- $\ulcorner a \urcorner \doteq \ulcorner b \urcorner$ is a code for $a =_{\mathcal{A}} b$;
- $\mathcal{S} \vdash \ulcorner a \urcorner \doteq \ulcorner b \urcorner$ if and only if $\mathcal{S} \vdash s \doteq t$ for some code $s \doteq t$ of $a =_{\mathcal{A}} b$.

The conclusion is that one can define an algorithm which performs the optimal choice for the NPS. In the sequel, it is understood that the NPS is always applied for such an optimal choice.

3.3.2 The Naming Principle for Extensionality.

The aim of the naming principle for extensionality is to recast a local equation $a =_{\mathcal{A}} b$ into an equation $[\![t]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![t]\!]^{\mathcal{A}}_{\beta}$, where

- t is a term of the theory T,
- $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,
- $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$.

In the second form, the equation can be immediately deduced from $\alpha x =_{\mathcal{A}} \beta x$ for all $x \in \mathsf{var}(t)$. As for the NPS, the method is only semi-complete. Yet it is a very important tool for formal proof development. Indeed, the standard representation of sets in most type systems uses the so-called setoids; consequently all the reasoning takes place with book equalities and extensionality matters do come up very often. As for the NPS, the NPE can be applied optimally. Indeed, one can find for every equational problem $a =_{\mathcal{A}} b$ a term t (the *optimal code* for NPE) such that

- there exist two assignments α and β such that $[\![t]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$ and $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$;
- for every term t' and assignments δ and γ such that $[\![t']\!]^{\mathcal{A}}_{\delta} \twoheadrightarrow a$ and $[\![t']\!]^{\mathcal{A}}_{\gamma} \twoheadrightarrow b$, there exists a substitution θ such that $\theta t' \twoheadrightarrow t$.

Note that it is possible to extend the naming principle for extensionality to formulae. Details will appear in [25].

3.3.3 Combining Both Principles.

In the previous subsections, we have considered two different naming principles which can be used to solve equational problems. However, the method that we have described disregards the possibility of using assumptions present in the context. In fact, the NPS is too weak to be useful in this more general case. For example, if one has to prove in a monoid M that

$$(a \circ b) \circ c =_{\mathcal{A}} a' \circ (b \circ c) \tag{2}$$

for some elements a, a', b and c of M such that $a =_{\mathcal{A}} a'$, the NPS reduces the problem to

$$[\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\gamma} =_{\mathcal{A}} [\![x' \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\gamma} \tag{3}$$

for a suitable assignment γ. Moreover, one cannot invoke the NPE principle to reduce equation (3) further. However, one can combine the NPS and the NPE to obtain a powerful naming principle (NPSE) which can be used to solve equational problems in a context. This new principle takes as input an equational problem $a =_{\mathcal{A}} b$ and returns as output an equation $[\![s]\!]^{\mathcal{A}}_{\alpha} = [\![t]\!]^{\mathcal{A}}_{\beta}$ where s and t are two terms s and t and α and β are two assignments such that

- $[\![s]\!]^{\mathcal{A}}_{\alpha} \twoheadrightarrow a$,
- $[\![t]\!]^{\mathcal{A}}_{\beta} \twoheadrightarrow b$,
- $\mathsf{dom}\ \alpha = \mathsf{dom}\ \beta$ and $\alpha x = \beta x$ for every $x \in \mathsf{dom}\ \alpha$.

As for the NPS, the equation follows from $\mathcal{S} \vdash s \doteq t$. With this new principle, equation (2) can be reduced to $[\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\alpha} =_{\mathcal{A}} [\![x \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\beta}$ and $\alpha x = \beta x$ for suitable α and β. This shows that the NPSE is stronger than the combination of the NPS and the NPE. However, it is difficult to find an optimal use of the NPSE for obvious reasons. Fortunately, one can recover the power of the NPSE from the NPS by grafting a simple procedure on top of the NPE. The procedure, called *collapsing procedure* (or CP for short),

- takes as input a problem of the form $[\![s]\!]^{\mathcal{A}}_{\alpha} = [\![t]\!]^{\mathcal{A}}_{\alpha}$ and two variables x and y in the domain of α,
- returns as output the problems $[\![s[y/x]]\!]^{\mathcal{A}}_{\alpha} = [\![t[y/x]]\!]^{\mathcal{A}}_{\alpha}$ and $\alpha x = \alpha y$.

The benefits of the CP are similar to those of the NPSE. For example, the CP can be called to reduce equation (3) into the two problems

$$\begin{aligned} [\![(x \cdot y) \cdot z]\!]^{\mathcal{A}}_{\gamma} &=_{\mathcal{A}} [\![x \cdot (y \cdot z)]\!]^{\mathcal{A}}_{\gamma} \\ \gamma x &=_{\mathcal{A}} \gamma x' \ . \end{aligned}$$

The CP provides an easy means to make use of the optimal naming of an equational problem via the NPS. Unfortunately, there does not seem to be any obvious counterpart for making use of the optimal naming of an equational problem via the NPE[4].

4 Oracle Types

4.1 How to Automate Equational Reasoning?

As mentioned earlier, the naming principles do not solve equational problems. A naming principle is a special kind of conversion rule which recasts an equational

[4] Consider a monoid H and three elements a, b, c of H such that $a \circ b =_H a' \circ b'$. The optimal use of the NPE on

$$(a \circ b) \circ c =_H (a' \circ b') \circ c$$

will yield the two subproblems $a =_H a'$ and $b =_H b'$. Indeed, the NPE will be applied with $(x \circ y) \circ z$ as code whereas it would have been better to take $x \circ y$ as code.

problem into a specific form. Here these specific forms are local satisfiability and extensionality problems. The point is the naming principles make apparent terms of an equational theory. In the special case where we look at a local satisfiability problem, the equational problem will become of the form $[\![s]\!]_\alpha^\mathcal{A} =_\mathcal{A} [\![t]\!]_\alpha^\mathcal{A}$. By the soundness/completeness theorem, the equality is a consequence of $\mathcal{S} \vdash s \doteq t$. Reducing an equational problem to a problem of the form $\mathcal{S} \vdash s \doteq t$ is useful because we dispose of techniques to determine whether an equation is in the deductive closure of an equational theory:

using computer algebra systems. Current computer algebra systems are excellent at equational reasoning. They have various clever algorithms to compute all kinds of equations at a symbolic (syntactical) level. We could use such a system to compute $s \doteq t$ and, if this succeeds, we let our proof checker assume the statement as an axiom. This is what we call the *external believing* way.

using term-rewriting. Another technique to check $\mathcal{S} \vdash s \doteq t$ is of course term-rewriting: if $\mathcal{S}$ can be completed into a confluent and terminating term-rewriting system $\mathcal{R}$, we can look at the normal form of s and t with respect to the completion of $\mathcal{S}$. For such theories, equational reasoning can be partially automated by using the naming principle and importing in some way term-rewriting into the type theory as done for example in [9]. We call this method the *internal believing* way, because the problem is solved without any outside help. This is the method proposed in this paper.

the autarkic way. We might want to define a map nf which assigns to every term its normal form in $\mathcal{R}$ and to show that for every term t and assignment α, we have $[\![t]\!]_\alpha^\mathcal{A} =_{\beta\iota} [\![\mathsf{nf}\ t]\!]_\alpha^\mathcal{A}$. In order to check $s \doteq t$, we just have to verify $(\mathtt{nf}\ s) = (\mathtt{nf}\ t)$, where $=$ denotes Leibniz equality. This comes down to a reflexivity test. This method is called the *autarkic* way because it does not involve any change to the type theory or the proof-checker. It must be said that this method seems currently too inefficient to be used in practice.

Most proposals in the literature opt for the external believing approach [2, 15, 17]. Indeed, the external believing way has an obvious advantage: hybrid systems offer a shortcut to integrate term-rewriting in proof checking. However, the approach has two disadvantages:

- proof checkers are based on well-understood languages whose logical and computational status are well understood. It is not always the case for computer algebra systems.
- proof checkers generate from scripts proof-objects; if the computer algebra system is used as an oracle, then all calculations performed by the computer algebra system have to be taken as axioms by the proof checker.

Such a process threatens the reliability of the hybrid system[5].

One can remedy to these two problems by using the computer algebra system not as an oracle but as a guide, as done in [15]. In this case, the answer of the computer algebra system is used to solve an equation. We call this method the *skeptic* way because the proof-checker does not trust the computer algebra system. This technique is superior over the external believing one in that it eliminates the holes in the proof-terms. Moreover, the problem of the reliability of the computer algebra system is circumvented. However the skeptic way seems unfeasable in a proof-checker such as Lego because of the absence of tactics.

4.2 The Internal Believing Approach via Oracle Types

In this section, we introduce oracle types. The formalism, which is based on algebraic, inductive and quotient types, is well-suited for the introduction of canonical term-rewriting systems. We refer the reader to [7] for a general scheme for oracle types and focus on a specific example of oracle type used to solve equational problems for groups. It consists of two types:

- an inductive type $\underline{G}$ corresponding to the set of terms of the signature of groups,
- the quotient G of $\underline{G}$ by the deductive closure of the theory of groups; G is defined as an algebraic type, i.e. equality between inhabitants of G is forced by the rewrite rules.

Both types are related by a map $[\_] : \underline{G} \to G$ which assigns to every term its equivalence class under the provability relation. There is an axiom to reflect the universal property of quotients as it is used in the completeness theorem: an equation $s \doteq t$ holds in every group if $[s] = [t]$. If we work in ECC [19], the rules are:

$$\frac{}{\vdash \underline{G} : \Box_0} \qquad \frac{}{\vdash \underline{e} : \underline{G}} \qquad \frac{}{\vdash \underline{i} : \underline{G} \to \underline{G}} \qquad \frac{}{\vdash \underline{o} : \underline{G} \to \underline{G} \to \underline{G}}$$

$$\frac{}{\vdash G : \Box_0} \qquad \frac{}{\vdash e : G} \qquad \frac{}{\vdash i : G \to G} \qquad \frac{}{\vdash o : G \to G \to G}$$

$$\frac{}{\vdash \underline{a} : \mathbb{N} \to \underline{G}} \qquad \frac{}{\vdash a : \mathbb{N} \to G} \qquad \frac{\Gamma \vdash b : \underline{G}}{\Gamma \vdash [b] : G} \qquad \frac{\Gamma \vdash p : [a] = [b]}{\Gamma \vdash \texttt{noconf}\ p : a =_{\underline{G}} b}$$

$$\frac{\Gamma \vdash C : \Box_0 \quad \begin{array}{c}\Gamma \vdash f_e : C \\ \Gamma \vdash f_a : \mathbb{N} \to C\end{array} \quad \begin{array}{c}\Gamma \vdash f_i : \underline{G} \to C \to C \\ \Gamma \vdash f_o : \underline{G} \to \underline{G} \to C \to C \to C\end{array}}{\epsilon^C\ [f_a, f_e, f_i, f_o] : \underline{G} \to C}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : s}{\Gamma \vdash a : B} \qquad \text{if } A \to_{\beta\chi\iota\rho} B \text{ or } B \to_{\beta\chi\iota\rho} A$$

[5]Sometimes the user has to make sure that the necessary side conditions are satisfied. For example, several computer algebra systems will state that $(\sqrt{x})^2$ equals x, without bothering about the condition that $x \geq 0$.

where $=_{\underline{G}}$ is the (impredicatively defined) deductive closure of the theory of groups, $[\_]$ is a new constructor and $\mathbb{N}$ are the inductively defined natural numbers. The computational content of the system is given by β-reduction and the following reduction relations:

– ι-reduction; let $\vec{f} = (f_a, f_e, f_i, f_o)$. The rules are

$$\begin{array}{rcl} \epsilon^C\ [\vec{f}]\ (a\ i) & \to_\iota & f_a\ i \\ \epsilon^C\ [\vec{f}]\ e & \to_\iota & f_e \\ \epsilon^C\ [\vec{f}]\ (i\ x) & \to_\iota & f_i\ x\ (\epsilon^C\ [\vec{f}]\ x) \\ \epsilon^C\ [\vec{f}]\ (o\ x\ y) & \to_\iota & f_o\ x\ y\ (\epsilon^C\ [\vec{f}]\ x)\ (\epsilon^C\ [\vec{f}]\ y) \end{array}$$

– ρ-reduction; the rules correspond to the Knuth-Bendix completion of the axioms of groups:

$$\begin{array}{rclcrcl} o\ e\ x & \to_\rho & x & \quad & i\ e & \to_\rho & e \\ o\ x\ e & \to_\rho & x & & o\ (o\ x\ (i\ y))\ y & \to_\rho & x \\ o\ x\ (o\ y\ z) & \to_\rho & o\ (o\ x\ y)\ z & & o\ (o\ x\ y)\ (i\ y) & \to_\rho & x \\ o\ (i\ x)\ x & \to_\rho & e & & i\ (i\ x) & \to_\rho & x \\ o\ x\ (i\ x) & \to_\rho & e & & i\ (o\ x\ y) & \to_\rho & o\ (i\ y)\ (i\ x) \end{array}$$

– χ-reduction; for every $x, y : \underline{G}$,

$$\begin{array}{rcl} [\underline{o}\ x\ y] & \to_\chi & o\ [x]\ [y] \\ [\underline{i}\ x] & \to_\chi & i\ [x] \\ [\underline{e}] & \to_\chi & e \end{array}$$

Note that the rules we present here are in fact a subset of the usual rules for *congruence* types.

5 Formalization in Lego

Type theory based proof checkers such as Alf, Coq and Lego are expressive enough for the two-level approach described above to be developed within the system itself. We present an implementation of the two-level approach in Lego. The reason to choose Lego is that it allows for the user to input its own rewrite rules, thus offering the possibility to implement oracle types.

5.1 Formalization of Equational Logic

Formalizing equational logic in Lego is relatively easy. There are no major difficulties in developing the whole theory along the lines of section 2. We can define a type of signatures as

```
Signature == <T:Type> T -> nat
```

where nat is the inductively defined type of natural numbers. The set of (n-tuples of) terms over a set of variables is defined as an inductive type. Equations are defined as pairs of terms and equational theories as signatures together with a predicate over the type of equations. One can even formalize the deductive closure of a set of equations by formalizing first the notion of simultaneous substitution. It is equally easy to define the semantics of equational logic. The definitions of algebra, assignment, satisfaction and model are immediate adaptations of the definitions introduced in section 3. See [25] for a more detailed presentation of our implementation of universal algebra in Lego.

5.2 Formalization of the Naming Principles

Lego does not offer support for the naming and extensionality principles[6]. Yet they are special instances of conversion rules, so they can be performed manually using the Equiv command. We present three examples, one using the NPE, a second using the CP and the third one using the NPS. These examples are meant to give an idea of the method used. To understand them fully, the reader should read first Appendix B. In each case, the proofs turn out to be remarkably short. Note that in our implementation we did not use (nor need) specifications of equational theories.

First, we give an example where the NPE is used to solve an equational problem. Here G is an algebra for the signature of groups, obj G is an element of its carrier, times is the multiplication on G and inv is the inverse on G. TIMES and INV are function symbols of the signature of groups. int is the interpretation function which, given an assignment rho, assigns a symbol of the signature to an element of G whose set of variables is contained in the domain of rho. Note that [x:A]b stands for $\lambda x{:}A.b$, {x:A}B for $\Pi x{:}A.B$, <x:A>B for $\Sigma x{:}A.B$, Set stands for the type of setoids, Eq for the equality of a Set, el for the elements of a Set, obj for the elements of a model and Q is Leibniz equality.

```
Lego> Goal {x,y,z:obj G} (Eq x y) ->
                         (Eq (times (inv x) z) (times (inv y) z));
Goal
  ?0 : {x,y,z:obj G}(Eq x y)->Eq (times (inv x) z) (times (inv y) z)
Lego> intros;
intros (4)
  x : obj G
  y : obj G
  z : obj G
  H : Eq x y
  ?1 : Eq (times (inv x) z) (times (inv y) z)
Lego> rho == necons x (necons y (base z));
defn  rho = necons x (necons y (base z))
      rho : nelist (obj G)
```

[6] An extension of the Lego system is proposed in [6] to solve this problem.

```
Lego> t   == TIMES (INV (VAR ZeroN)) (VAR TwoN);
defn  t = TIMES (INV (VAR ZeroN)) (VAR TwoN)
      t : termGr
Lego> u   == TIMES (INV (VAR  OneN)) (VAR TwoN);
defn  u = TIMES (INV (VAR OneN)) (VAR TwoN)
      u : termGr
Lego> Equiv Eq (int G rho t) (int G rho u);
Equiv
  ?2 : Eq (int G rho t) (int G rho u)
Lego> Refine SubstitutionLemma G ZeroN;
Refine by  SubstitutionLemma G ZeroN
  ?9 : Eq (int G rho (TFV sig ZeroN)) (int G rho (VAR OneN))
Lego> Refine H;
Refine by  H
Discharge..  rho H z y x
*** QED ***
```

Note that the NPE yields the goal `?2`. The `SubstitutionLemma` is used to obtain `?9` is a specific instance of the compatibility lemma. The next example uses the CP procedure. Here we are working in a context in which `times_assoc` states that `times` is associative. The CP procedure is called by the term `CP`.

```
Lego> Goal {a,b,b',c:obj G} (Eq b b') ->
          Eq (times a (times b c)) (times (times a b') c);
Goal
  ?0 : {a,b,b',c:obj G} (Eq b b') ->
                        (Eq (times a (times b c)) (times (times a b') c)
Lego> intros;
intros (5)
  a : obj G
  b : obj G
  b' : obj G
  c : obj G
  H : Eq b b'
  ?1 :Eq (times a (times b c)) (times (times a b') c)
Lego> rho == necons a (necons b (necons b' (base c)));
defn  rho = necons a (necons b (necons b' (base c)))
      rho : nelist (obj G)
Lego> t   == TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR ThreeN));
defn  t = TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR ThreeN))
      t : termGr
Lego> u   == TIMES (TIMES (VAR ZeroN) (VAR TwoN)) (VAR ThreeN);
defn  u = TIMES (TIMES (VAR ZeroN) (VAR TwoN)) (VAR ThreeN)
      u : termGr
Lego> Equiv Eq (int G rho t) (int G rho u);
Equiv
  ?1 : Eq (int G rho u) (int G rho u)
Lego> Refine CP G OneN (VAR TwoN);
Refine by  CP G OneN (VAR TwoN)
  ?9 : Eq (int G rho (TFV sig OneN)) (int G rho (VAR TwoN))
```

```
  ?10 : Eq (int G rho (Subst t OneN (VAR TwoN)))
           (int G rho (Subst u OneN (VAR TwoN)))
Lego> Refine H;
Refine by  H
?10 : ...
Lego> Refine times_assoc;
Refine by  times_assoc
Discharge..  rho H c b' b a
*** QED ***
```

The final example uses the NPS. Oracle types are used to give a short proof of an equality on groups. In the sequel, `Q_refl` is a proof of the reflexivity of Leibniz equality, `comm` and `conj` respectively denote the commutator and the conjugate of two elements. For comparison, we have included a traditional proof of this fact in appendix B.

```
Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  rho == necons x (necons y (base z));
  t   == CONJ (COMM (VAR ZeroN) (VAR OneN)) (VAR TwoN);
  u   == COMM (CONJ (VAR ZeroN) (VAR TwoN)) (CONJ (VAR OneN) (VAR TwoN));
  Equiv Eq (int G rho t) (int G rho u);
  Refine Soundness;
  Refine Q_refl;
Save comm_conj;
```

6 Conclusions

We have developed a simple, flexible and rather efficient method to solve equational problems in type theory. The main ingredients of our method are a two-level formalization of universal algebra based on oracle types. The approach chosen in this paper is also intimately related to the design of hybrid systems and can be seen as an attempt to lay the foundations for a theoretical understanding of the interaction between proof checkers and computer algebra systems. In the future, it seems worthwhile to try to extend the framework to equational theories which do not yield a confluent terminating term-rewriting system. A longer term goal related to this research is the understanding of computer algebra algorithms. A full understanding of their nature as term-rewriting systems is necessary to see whether a type system with (a reasonable variant of) oracle types can provide a theoretical framework in which the integration of proof checkers and computer algebra systems can be justified.

References

[1] A. Bailey. *Representing algebra in Lego*, M.Sc. thesis, University of Edinburgh, October 1993.

[2] C. Ballarins, K. Homann and J. Calmet. *Theorems and algorithms: an interface between Maple and Isabelle*, in the proceedings of ISSAC'95.

[3] H.P. Barendregt. *Typed λ-calculi*, Handbook of logic in computer science, Abramsky and *al* eds, OUP 1992.

[4] G. Barthe. *Towards a mathematical vernacular*, manuscript, presented at the HISC workshop, Amsterdam, March 1994.

[5] G. Barthe. *Formalising mathematics in type theory: fundamentals and case studies*, manuscript, June 1994, submitted for publication.

[6] G. Barthe and H. Elbers. *Towards lean proof checking*, to appear in the proceedings of DISCO'96, Lecture Notes in Computer Science, Springer-Verlag, 1996. An extended version will appear as a CWI technical report.

[7] G. Barthe and H. Geuvers. *Congruence types*, to appear in the proceedings of CSL'95, 1995.

[8] G. Barthe, M. Ruys and H. Barendregt. *A two-level approach towards lean proof-checking*, to appear as a CWI technical report, 1996.

[9] V. Breazu-Tannen. *Combining algebra and higher-order types*, in the proceedings of LICS'88, pp 82-90, IEEE, 1988.

[10] P. Cohn. *Universal algebra*, Mathematics and its Applications, Vol. 6, D. Reidel, 1981.

[11] R. Constable. *Metalevel Programming in Constructive Type Theory*, Logic and Algebra of Specification, F. Bauer and *al* eds, NATO Asi Series, 1994.

[12] R. Constable and *al*. *Implementing mathematics with the NuPrl proof development system*, Prentice Hall, 1986.

[13] G. Dowek and *al*. *The Coq proof assistant user's guide* Technical Report, INRIA, November 1993.

[14] H. Elbers. *A machine-assisted construction of the real numbers*, M.Sc. thesis, University of Nijmegen, September 1993.

[15] J. Harrison and L. Théry. *Extending the HOL theorem prover with a computer algebra system to reason about the reals*, in proceedings of HOL'93, LNCS, 1993.

[16] D. Howe. *Automating reasoning in an implementation of constructive type theory*, Ph.D. thesis, Cornell University, 1988.

[17] P. Jackson. *Exploring abstract algebra in constructive type theory*, in the proceedings of CADE-12, LNAI 814, June 1994.

[18] J.W. Klop. *Term-rewriting systems*, in Handbook of logic in computer science (volume 2), Abramsky and *al* eds, OUP 1992.

[19] Z. Luo. *Computation and reasoning: a type theory for computer science*, OUP, 1994.

[20] Z. Luo and R. Pollack. *LEGO proof development system: user's manual*, Technical Report, University of Edinburgh, May 1992.

[21] L. Magnusson and B. Nordström. *The Alf proof editor and its proof engine*, in the proceedings of Types for Proofs and Programs, LNCS 806, May 1993.

[22] P. Martin-Löf. *An intuitionistic theory of types*, Bibliopolis, 1984.

[23] R. Nederpelt and *al. Selected papers on AUTOMATH*, North-Holland, 1994.

[24] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's type theory*, OUP, 1990.

[25] M.P.J. Ruys. Ph.D. thesis, University of Nijmegen, forthcoming (1996).

A Formalization of Universal Algebra

This appendix contains a pruned file of our implementation in Lego. All proofs have been removed. The complete set of files can be obtained via WWW at `http://www.cs.kun.nl/fnds/papers/two-level.shtml`[7]. The reader is referred to [25] for an elaboration in greater detail on formalizing mathematics (and universal algebras) in type theory.

```
(* ====================================================================
   Syntax
*)

(* A signature is a set S of function symbols and an arity map. *)

[Signature = <S:SET> S -> nat];
[FuncSymb  [s:Signature] : SET                 = s.1]
[FuncArity [s:Signature] : (FuncSymb s) -> nat = s.2];

(* Let s be a signature. Define n-tuples of terms. Define term. *)

[s : Signature];

Inductive [terms : nat -> SET]
Constructors [TFV  : nat -> (terms OneN)]
             [TFC  : {c:FuncSymb s} (terms (FuncArity c)) -> terms OneN]
             [tnil : terms ZeroN]
             [tconc: {n:nat}(terms OneN) -> (terms n) -> terms (succ n)];

[term : SET = terms OneN];
```

[7] Anonymous ftp from `ftp.cs.kun.nl` in the directory `/pub/CSI/CompMath.Found/two-level` is also possible.

```
(* terms and term are made into the setoids using Leibniz equality. *)

[Terms [n:nat] : Set = QSet (terms n)];
[Term          : Set = QSet term     ];

(* Let s be a signature. Define simultaneous and normal substitution. *)

Goal SimSubst: {rho:nelist (prod nat term)}{n:nat}(terms n) -> (terms n);
Save;

Goal Subst_n : {n:nat} (terms n) -> nat -> term -> (terms n);
Save;

Goal Subst : term -> nat -> term -> term;
Save;

Discharge s;

(* ======================================================================
   Semantics
*)

[Algebra [s:Signature] = <A:Set> {c:FuncSymb s} nFunc A (FuncArity c)];

[sig : Signature] [A : Algebra sig];

[car           :    Set                          = A.1]
[obj           :    SET                          = el car]
[assignment    :    SET                          = nelist obj]
[Assignment    :    Set                          = neList car];

(* Define the interpretation of tuples of terms with respect to an
   assignment. *)

Goal int_n : {n:nat} assignment -> (terms sig n) -> (product obj n);
Save;

(* Show int_n preserves equality. *)

Goal {n:nat} extensional2 Assignment (Terms sig n) (Product car n)
     (int_n n);
Save int_n_exten;

(* Define the interpretation of terms with respect to an assignment. *)

Goal int : assignment -> (term sig) -> obj;
Save;

(* Show int preserves equality. *)
```

```
Goal extensional2 Assignment (Term sig) car int;
Save int_exten;

(* Prove the Compatibility lemma: rho = rho' -> [t]_rho = [t]_rho' *)

Goal {rho,rho':el Assignment} (Eq rho rho') ->
     {t:term sig} Eq (int rho t) (int rho' t);
Save CompAss;

(* Prove [VAR x] = [u]  -> [t] = [t[x:=u]] *)

Goal {t:term sig}{rho:asigsnment}{x:nat}{u:term sig}
     (Eq (int rho (TFV sig x)) (int rho u)) ->
     (Eq (int rho t) (int rho (Subst t x u)));
Save SubstitutionLemma;

(* Prove [VAR y] = [u]  ->  [s[y:=u]] = [t[y:=u]]  -> [s] = [t] *)

Goal {s,t:term sig} {rho:assignment} {y:nat} {u:term sig}
     (Eq (int rho (TFV sig y)) (int rho u)) ->
     (Eq (int rho (Subst s y u)) (int rho (Subst t y u))) ->
     (Eq (int rho s) (int rho t));
Save CP;

Discharge sig;
```

B Examples

This appendix contains examples of equational problems solved using our approach. To keep the presentation simple, we introduce the group axioms without using an equational theory. Note that because of the two-level approach, the number of Lego commands of the proof comm_conj is very small (in essence only four). This in contrast to the traditional proof comm_conj_hand. Because of a lot of applications of the transitivity of equality and the group axioms, the proof explodes up to a few pages of Lego commands.

```
Module Examples Import syntax semantics;

(* Define the signature and the terms of a Group. *)

[sigGr  : Signature                 = ...]
[termGr : SET                       = term sigGr]
[VAR    : nat -> termGr             = TFV sigGr]
[ONE    : termGr                    = ...]
[INV    : termGr -> termGr          = ...]
[TIMES  : termGr-> termGr -> termGr = ...];
[DIV    : termGr-> termGr -> termGr = ...];
```

```
(* Let G be a group, satisfying the group axioms. *)

[G : Algebra sigGr];

[One   : el (car G)                    = ...]
[Inv   : Fun (car G) (car G)           = ...]
[Times : Fun2 (car G) (car G) (car G)  = ...]
[one   : obj G                         = ...]
[inv   : (obj G) -> (obj G)            = ...]
[times : (obj G) -> (obj G) -> (obj G) = ...];

[One_ident   : Identity Times One   ]
[Inv_invers  : Inverse Times One Inv]
[Times_assoc : Associative Times    ];

(* Show y = z  ->  z ((x/y) y) = z ((x/z) z) *)

Goal {x,y,z:obj G} (Eq y z) -> Eq (times (times y (times x (inv y))) z)
                                  (times (times z (times x (inv z))) z);
  intros;
  rho == necons x (necons y (base z));
  t  == TIMES (TIMES (VAR OneN) (DIV (VAR ZeroN) (VAR OneN))) (VAR TwoN);
  u  == TIMES (TIMES (VAR TwoN) (DIV (VAR ZeroN) (VAR TwoN))) (VAR TwoN);
  Equiv Eq (int G rho t) (int G rho u);
  Refine SubstitutionLemma G OneN;
  Refine H;
Save Example_1;

(* Show b = b'  ->  a (b c) = (a b') c *)

Goal {a,b,c,d:obj G} (Eq b d) -> Eq (times a (times b c))
                                    (times (times a d) c);
  intros;
  rho == necons a (necons b (necons c (base d)));
  t   == TIMES (VAR ZeroN) (TIMES (VAR OneN) (VAR TwoN));
  u   == TIMES (TIMES (VAR ZeroN) (VAR ThreeN)) (VAR TwoN);
  Equiv Eq (int G rho t) (int G rho u);
  Refine CP G OneN (VAR TwoN);
  Refine H;
  Refine Times_assoc;
Save Example_2;

(* ====================================================================
   Use Oracle Types to implement term rewriting.                      *)

[FreeGroup : SET];

[varFg   : nat -> FreeGroup];
[oneFg   : FreeGroup];
```

```
[invFg   : FreeGroup -> FreeGroup];
[timesFg : FreeGroup -> FreeGroup -> FreeGroup];

(* Define the Knuth-Bendix completion of the group equations. *)

[ [x,y,z : FreeGroup]

   timesFg oneFg x                 ==> x
|| timesFg x oneFg                 ==> x
|| timesFg (invFg x) x             ==> oneFg
|| timesFg x (invFg x)             ==> oneFg
|| invFg oneFg                     ==> oneFg
|| timesFg (timesFg x (invFg z)) z ==> x
|| timesFg (timesFg x y) (invFg y) ==> x
|| timesFg x (timesFg y z)         ==> timesFg (timesFg x y) z
|| invFg (invFg z)                 ==> z
|| invFg (timesFg z y)             ==> timesFg (invFg y) (invFg z)
];

[class : termGr -> FreeGroup = ...];

[Soundness : {s,t:termGr} {rho:el (Assignment G)}
             (Q (class s) (class t)) -> Eq (int G rho s) (int G rho t)];

(* ----------------------------------------------------------------------
   The conjugate of a commutator equals the commutator of the conjugates.

   Define the commutator [x,y] == (x y)/(y x)
   and the conjugate      x*y  == y (x/y)                              *)

[comm [x,y : obj G]  : obj G  = times (times x y) (inv (times y x))]
[COMM [x,y : termGr] : termGr = DIV (TIMES x y) (TIMES y x)]
[conj [x,y : obj G]  : obj G  = times y (times x (inv y))]
[CONJ [x,y : termGr] : termGr = TIMES y (TIMES x (INV y))];

(* Show [x,y]*z = [x*z,y*z] using the two-level approach. *)

Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  rho == necons x (necons y (base z));
  t   == CONJ (COMM (VAR OneN) (VAR OneN)) (VAR TwoN);
  u   == COMM (CONJ (VAR OneN) (VAR TwoN)) (CONJ (VAR OneN) (VAR TwoN));
  Equiv Eq (int G rho t) (int G rho u);
  Refine Soundness;
  Refine Q_refl;
Save comm_conj;

(* ----------------------------------------------------------------------
   Proof the last lemma again on the traditional way.
   First show x^{-1}^{-1} = x.                                         *)
```

```
Goal Involutive Inv;
  Intros x;
  Refine Eq_trans (times one (inv (inv x)));
    Refine Eq_sym; Refine fst One_ident;
  Refine Eq_trans (times (times x (inv x)) (inv (inv x)));
    Refine exten2 Times ??.Eq_refl; Refine Eq_sym; Refine snd Inv_invers;
  Refine Eq_trans (times x (times (inv x) (inv (inv x))));
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times x one);
    Refine exten2 Times ?.Eq_refl; Refine snd Inv_invers;
  Refine snd One_ident;
Save Inv_invol;

(* Show (x y)^{-1} = y^{-1} x^{-1} *)

Goal {x,y:obj G} Eq (inv (times x y)) (times (inv y) (inv x));
  intros;
  Refine Eq_sym;
  Refine Eq_trans (times (times (inv y) (inv x)) one);
    Refine Eq_sym; Refine snd One_ident;
  Refine Eq_trans (times (times (inv y) (inv x))
                        (times (times x y) (inv (times x y))));
    Refine exten2 Times ?.Eq_refl; Refine Eq_sym; Refine snd Inv_invers;
  Refine Eq_trans (times (times (times (inv y) (inv x)) (times x y))
                        (inv (times x y)));
    Refine Times_assoc;
  Refine Eq_trans (times one (inv (times x y)));
    Refine +1 fst One_ident;
  Refine exten2 ? ? ?.Eq_refl;
  Refine Eq_trans (times (inv y) (times (inv x) (times x y)));
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times (inv y) y);
    Refine +1 fst Inv_invers;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times (times (inv x) x) y);
    Refine Times_assoc;
  Refine Eq_trans (times one y);
    Refine exten2 Times ? ?.Eq_refl; Refine fst Inv_invers;
  Refine fst One_ident;
Save Times_Inv;

(* Show (x*z)(y*z) = (x y)*z *)

Goal {x,y,z:obj G} Eq (times (conj x z) (conj y z)) (conj (times x y) z);
  intros;
  Refine Eq_trans (times z (times (times x (inv z)) (conj y z)));
    Refine Eq_sym; Refine Times_assoc;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times x (times (inv z) (conj y z)));
```

```
    Refine Eq_sym; Refine Times_assoc;
  Refine Eq_trans (times x (times y (inv z)));
    Refine +1 Times_assoc;
  Refine exten2 Times ?.Eq_refl;
  Refine Eq_trans (times (times (inv z) z) (times y (inv z)));
    Refine Times_assoc;
  Refine Eq_trans (times one (times y (inv z)));
    Refine +1 fst One_ident;
  Refine exten2 Times ? ?.Eq_refl;
  Refine fst Inv_invers;
Save Times_Conj;

(* Show (x*y)^{-1} = x^{-1}*y *)

Goal {x,y:obj G} Eq (inv (conj x y)) (conj (inv x) y);
  intros;
  Refine Eq_trans (times (inv (times x (inv y))) (inv y));
    Refine Times_Inv;
  Refine Eq_trans (times (times y (inv x)) (inv y));
    Refine +1 Eq_sym; Refine +1 Times_assoc;
  Refine exten2 Times ? ?.Eq_refl;
  Refine Eq_trans (times (inv (inv y)) (inv x));
    Refine Times_Inv;
  Refine exten2 Times ? ?.Eq_refl;
  Refine Inv_invol;
Save Inv_conj;

(* And finally, show [x,y]*z = [x*z,y*z] *)

Goal {x,y,z:obj G} Eq (conj (comm x y) z) (comm (conj x z) (conj y z));
  intros;
  Refine Eq_sym;
  Refine Eq_trans (times (conj (times x y) z) (conj (inv(times x y)) z));
    Refine +1 Times_Conj;
  Refine exten2 Times; Refine Times_Conj;
  Refine Eq_trans (inv (conj (times x y) z));
    Refine exten Inv; Refine Times_Conj;
  Refine Inv_conj;
Save comm_conj_hand;
```