Proofs by induction in equational theories with constructors

Gérard Huet, J.M. Hullot

INRIA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105 78150 Le Chesnay
France
Tél.:954 90 20

Rapports de Recherche

N° 28

PROOFS BY INDUCTION IN EQUATIONAL THEORIES WITH CONSTRUCTORS

Gérard HUET
Jean-Marie HULLOT

Août 1980

Proofs by Induction in Equational Theories with Constructors

Gérard HUET and Jean-Marie HULLOT

Résumé

Nous montrons comment faire des démonstrations (et des réfutations) d'identités dans l'algèbre initiale d'une variété equationnelle, par une simple extension de l'algorithme de complétion de Knuth et Bendix. Ceci nous permet de démontrer par des méthodes équationnelles des théorèmes dont la preuve nécessite d'ordinaire l'utilisation d'un principe de récurrence. Nous montrons des applications de cette méthode à des preuves de programmes calculant sur des structures de données récursives, et à des preuves de sommations algébriques. Ce travail étend et simplifie des résultats récents de Musser et de Goguen.

Abstract

We show how to prove (and disprove) theorems in the initial algebra of an equational variety by a simple extension of the Knuth-Bendix completion algorithm. This allows us to prove by purely equational reasoning theorems whose proof usually requires induction. We show applications of this method to proofs of programs computing over data structures, and to proofs of algebraic summation identities. This work extends and simplifies recent results of Musser and Goguen.

PROOFS BY INDUCTION IN EQUATIONAL THEORIES WITH CONSTRUCTORS

Gérard Huet and Jean-Marie Hullot

INRIA

Abstract

We show how to prove (and disprove) theorems in the initial algebra of an equational variety by a simple extension of the Knuth-Bendix completion algorithm. This allows us to prove by purely equational reasoning theorems whose proof usually requires induction. We show applications of this method to proofs of programs computing over data structures, and to proofs of algebraic summation identities. This work extends and simplifies recent results of Musser[15] and Goguen[6].

Introduction

We assume familiarity with the basic notions of equational logic and term rewriting systems. See for instance[11]. For simplicity of notation, we assume we have only one sort; all the results of this paper carry over to many-sorted theories without difficulty.

A set of equations $\mathcal{E}$ defines a *variety*, that is the class of algebras which are models of the equations considered as axioms. An equation $M = N$ is said to be valid in this variety if it is true in all these models. It is well known that this is equivalent to whether $M = N$ can be derived from $\mathcal{E}$, using instantiation and replacement of equals by equals. In the cases where $\mathcal{E}$ can be compiled into a *canonical term rewriting system* by the Knuth-Bendix completion algorithm[13], we can decide this problem by testing for identity the canonical forms of M and N.

Equations may also be used as definitions. This is frequent in computer science: programs written in applicative programming languages, abstract interpreter definitions and algebraic data type specifications are of this nature. In this framework, one has in mind a notion of standard model defined by these equations: the *initial algebra* defined by the set of equations. Now we have lost the nice completeness property of equational logic: an equation $M = N$ cannot in general be proved to be valid (or invalid) in the initial algebra by mere equational reasoning: some kind of induction is necessary.

However, Musser has recently shown an interesting theorem which may be roughly stated as follows: if the set of equations considered contains the axiomatization of an equality predicate, then an equation is valid in the initial algebra if and only if adding it as an axiom does not make the theory inconsistent (in the sense that true = false is derivable). This permits proofs (and disproofs) of equations without explicit induction. The method was simplified by Goguen[6] and Huet and Oppen[11].

We show in this paper that in the case where one considers inductive definitions over free algebras, and when the Knuth-Bendix completion algorithm converges, we can make these proofs by a very simple extension of the completion algorithm, and without the need of an equality axiomatization. We show how the method applies to proofs of simple properties and optimizations of primitive recursive programs over recursively defined data structures. The inductive completion algorithm defined in the paper generates implicitly the necessary instances of structural induction. The method generalizes to commutative-associative theories, and we show an application to proofs of algebraic summation identities.

1. A Principle of Definition

The key of our method consists in partitioning our function symbols between constructors and defined function symbols, and to express the necessary relationships between them via a principle of definition.

We assume given signature Σ. Every operator F in Σ is given with its arity. The signature Σ is partitioned as $\Sigma = C \uplus D$. We call operators in C the *constructors*, and members of D the *defined* operators. We assume there are at least two constructors (for instance, true and false).

This work was partially supported by AFOSR Contract F49620-79-C-0099, by ONR Contract N00014-75-C-0816 and by a fellowship from INRIA while the authors were visiting SRI International.

Let $\mathcal{T}$ be the set of terms constructed from operators in Σ and variables in a given denumerable set $\mathcal{V}$. We use $\mathcal{G}$ to denote the set of *ground* terms, i.e. containing no variables, and we assume $\mathcal{G}$ nonempty. Finally we denote by $\mathcal{GC}$ the set of ground terms formed solely from constructors.

Principle of Definition. Let $\mathcal{E}$ be a set of equations over Σ, $=_{\mathcal{E}}$ the corresponding congruence on $\mathcal{T}$. We say that $\mathcal{E}$ *defines* D *over* C if and only if for every M in $\mathcal{G}$ there exists a unique N in $\mathcal{GC}$ such that $M =_{\mathcal{E}} N$.

It is convenient to express our principle of definition as the conjunction of two properties:

(1) For every M in $\mathcal{G}$ there exists N in $\mathcal{GC}$ such that $M =_{\mathcal{E}} N$.

(2) For every M, N in $\mathcal{GC}$ we have $M =_{\mathcal{E}} N$ only if $M = N$.

When $\mathcal{E}$ satisfies (1), we shall use $\mathcal{GC}_{\mathcal{E}}[M]$, for M in $\mathcal{G}$, to denote any N in $\mathcal{GC}$ such that $M =_{\mathcal{E}} N$. Note that (1) implies that we have a constructor signature in the sense of Goguen[6]. If $\mathcal{E}$ satisfies (2) as well, $\mathcal{GC}_{\mathcal{E}}$ is a function, and then the set $\mathcal{GC}$ can easily be made into a Σ-algebra by associating with F of arity n the function $\lambda\, M_1, \ldots, M_n . \mathcal{GC}_{\mathcal{E}}[F(M_1, \ldots, M_n)]$. Moreover:

Lemma 1. *If $\mathcal{E}$ satisfies the principle of definition, the algebra $\mathcal{GC}$ is isomorphic to the initial algebra $I(\Sigma, \mathcal{E})$.*

Proof. Follows directly from the fact that the initial algebra is (isomorphic to) the quotient of $\mathcal{G}$ by $=_{\mathcal{E}}$. (See for instance[8]). ∎

2. Sufficient Conditions for Deciding the Definition Principle

Let us consider sufficient conditions for our principle of definition to hold. We shall from now on regard our sets of equations (when possible) as sets of (oriented) rewrite rules. We assume familiarity with the terminology of term rewriting systems[9,11]. In particular, we recall that a *canonical* term rewriting system is defined as being *confluent* (i.e. to have the Church-Rosser property) and *nœtherian* (i.e. all sequences of rewriting terminate).

Lemma 2. *Let $\mathcal{E}$ be such that it defines a nœtherian term rewriting system such that every term of the form $F(M_1, M_2, \ldots, M_n)$, with F in D and $M_1, \ldots, M_n$ in $\mathcal{GC}$, is reducible. Then $\mathcal{E}$ satisfies (1).*

Proof. Define $\mathcal{GC}_{\mathcal{E}}[M]$, for M in $\mathcal{G}$, as some $\mathcal{E}$-normal form of M. It is easy to show by structural induction that any such normal form must be in $\mathcal{GC}$. ∎

There are several ways to give effective conditions that are sufficient to entail the hypothesis of lemma 2. We shall propose here one such condition; we recommend skipping the details of the next definition on a first reading.

Definition. We define inductively what it means for a set $S = \{S_1, \ldots, S_p\}$ of k-tuples of terms $S_i = (S_i^1, \ldots, S_i^k)$ $(1 \leq i \leq p)$ to be *complete* for C. First we require every variable of S_i to occur in only one occurrence. Then either $k = 0$, and $S = \{\langle\rangle\}$, or else:

- either the set of $k-1$-tuples $\{(S_i^2, \ldots, S_i^k) \mid S_i^1 \in \mathcal{V}\}$ is complete,
- or else for every C in C, say of arity n, there is at least one S_i^1 with leading function symbol C, and the union of the two sets of $n + k - 1$-tuples $\{(P_1, \ldots, P_n, S_i^2, \ldots, S_i^k) \mid S_i^1 = C(P_1, \ldots, P_n)\}$ and $\{(x_1, \ldots, x_n, S_i^2, \ldots, S_i^k) \mid S_i^1 \in \mathcal{V}\}$ is complete, where the x_i's are new distinct variables not occurring in S.

Remark that this definition is well-founded, first on the number of function symbols contained in S, and second on k.

Example. With $C = \{\mathrm{S}, 0\}$, with S unary and 0 a constant, the following set is complete for C:
$\{\langle 0, \mathrm{S}(x)\rangle, \langle x, 0\rangle, \langle \mathrm{S}(x), \mathrm{S}(0)\rangle, \langle \mathrm{S}(x), \mathrm{S}(\mathrm{S}(y))\rangle\}$.

Lemma 3. *Let $S = \{S_1, \ldots, S_p\}$ be a set of k-tuples of terms complete for C. For every k-tuple of ground terms in $\mathcal{GC}$: $(M_1, \ldots, M_k)$ there exist n, with $1 \leq n \leq p$, and a substitution σ, such that for every ℓ, $1 \leq \ell \leq k$, we have $M_\ell = \sigma(S_n^\ell)$.*

Proof. Easy induction on the definition of complete. ∎

This lemma permits us to state a sufficient condition for property (1), which we shall use in practice:

Lemma 4. *Let $\mathcal{E}$ be a set of equations defining a nœtherian term rewriting system such that, for every F in D, there is in $\mathcal{E}$ a set of rewrite rules whose left-hand sides are of the form $F(S_i^1, \ldots, S_i^k)$, $(1 \leq i \leq p)$, and the set $\{S_1, \ldots, S_p\}$ is complete for C. Then $\mathcal{E}$ has property (1).*

Proof. It is easy to show, using lemma 3, that the assumption of the lemma implies that of lemma 2. ∎

Remark that if $\mathcal{E}$ is finite and known to be nœtherian, then the hypothesis of the lemma is a decidable condition. Actually, when giving the definition of F in D by cases on arguments constructed over C, one naturally gets complete sets of arguments.

Finally we state a trivial sufficient condition for property (2).

Lemma 5. *Let $\mathcal{E}$ be a set of equations defining a canonical term rewriting system such that every left-hand side is of the form $F(M_1, \ldots, M_n)$ with F in D. Then $\mathcal{E}$ has property (2).*

Proof. Since $\mathcal{E}$ is canonical, we have $M =_\mathcal{E} N$ if and only if $M\downarrow = N\downarrow$, where $M\downarrow$ denotes the canonical form of M obtained by an arbitrary terminating sequence of rewritings by rules in $\mathcal{E}$. If all left-hand sides of equations in $\mathcal{E}$ have their leading function symbol in D, we have $M\downarrow = M$ for every M in $\mathcal{GC}$. ∎

Putting together the two preceding lemmas gives a useful sufficient criterion for the principle of definition to hold. For instance, any set of primitive recursive definitions satisfies the hypotheses of lemmas 4 and 5.

Remark that if $\mathcal{E}$ obeys the hypothesis of lemma 5, then the converse of lemma 2 holds: $\mathcal{E}$ satisfies the definition principle if and only if $\mathcal{GC}$ is the set of $\mathcal{E}$-normal forms, and then $\mathcal{GC}_\mathcal{E}[M]$ is the canonical form of M defined by $\mathcal{E}$. However, the converse of lemma 4 may not hold, since property (1) may be the consequence of axioms in $\mathcal{E}$ whose left hand sides contain multiple occurrences of a variable. We shall return to this problem in section 5.

3. Structural Induction and the Principle of Definition

In this section, we shall show how our principle of definition permits us to prove and disprove properties of the standard model $I(\Sigma, \mathcal{E})$. The next lemma shows that the principle of definition is preserved by extension if and only if this extension is valid in the standard model.

Lemma 6. *Let $\mathcal{E}$ satisfy (1) above. Let $\mathcal{E}'$ be any set of Σ-equations such that $=_\mathcal{E}$ is contained in $=_{\mathcal{E}'}$. Then $\mathcal{E}'$ satisfies (2) if and only if:*

a) $\mathcal{E}$ satisfies (2), and

b) every equation of $\mathcal{E}'$ holds in $I(\Sigma, \mathcal{E})$.

Proof. Obviously, $\mathcal{E}'$ satisfies (1), and it satisfies (2) only if $\mathcal{E}$ does too.

$\Rightarrow$ Assume that $\mathcal{E}'$ satisfies (2) and that $M = N$ in $\mathcal{E}'$ does not hold in $I(\Sigma, \mathcal{E})$. This means that for some ground substitution σ we have $\sigma(M) \neq_\mathcal{E} \sigma(N)$. In particular we get $\mathcal{GC}_\mathcal{E}[\sigma(M)] \neq \mathcal{GC}_\mathcal{E}[\sigma(N)]$, although $\mathcal{GC}_\mathcal{E}[\sigma(M)] =_{\mathcal{E}'} \mathcal{GC}_\mathcal{E}[\sigma(N)]$, a contradiction with (2) for $\mathcal{E}'$.

$\Leftarrow$ If every equation of $\mathcal{E}'$ holds in $I(\Sigma, \mathcal{E})$, then for every M,N in $\mathcal{G}$ we have $M =_{\mathcal{E}'} N$ if and only if $M =_\mathcal{E} N$, and (2) for $\mathcal{E}'$ follows from (2) for $\mathcal{E}$. ∎

The next three lemmas give technical properties of equality in the standard model that are essential to the proof of our completion algorithm.

Lemma 7. *Let $M = C(M_1, \ldots, M_n)$, $N = C(N_1, \ldots, N_n)$, with C in C. Let $\mathcal{E}$ be a set of equations satisfying the principle of definition such that $M =_\mathcal{E} N$. Then $M_i = N_i$ holds in $I(\Sigma, \mathcal{E})$ for every i, $1 \leq i \leq n$.*

Proof. Let σ be an arbitrary ground substitution, and assume $M =_\mathcal{E} N$. We have

$$\sigma(M) = C(\sigma(M_1), \ldots, \sigma(M_n)) =_\mathcal{E} C(\sigma(N_1), \ldots, \sigma(N_n)) = \sigma(N)$$

and by (1) we get

$$C(\mathcal{GC}_\mathcal{E}[\sigma(M_1)], \ldots, \mathcal{GC}_\mathcal{E}[\sigma(M_n)]) =_\mathcal{E} C(\mathcal{GC}_\mathcal{E}[\sigma(N_1)], \ldots, \mathcal{GC}_\mathcal{E}[\sigma(N_n)])$$

which implies by (2) $\mathcal{GC}_\mathcal{E}[\sigma(M_i)] = \mathcal{GC}_\mathcal{E}[\sigma(N_i)]$ for every i, $1 \leq i \leq n$, and thus $\sigma(M_i) =_\mathcal{E} \sigma(N_i)$. Since this holds for every ground σ, we get that $M_i = N_i$ holds in $I(\Sigma, \mathcal{E})$. ∎

Corollary. *With M and N as above, let $\mathcal{E}$ containing $M = N$ and satisfying (1). Consider $\mathcal{E}' = \mathcal{E} - \{M = N\} \cup \{M_i = N_i \mid 1 \leq i \leq n\}$. Obviously $=_\mathcal{E}$ is contained in $=_{\mathcal{E}'}$. Now either $\mathcal{E}$ satisfies (2), in which case $I(\Sigma, \mathcal{E}') = I(\Sigma, \mathcal{E})$ by lemma 7, and $\mathcal{E}'$ satisfies (2) by lemma 6, or else $\mathcal{E}'$ does not satisfy (2).*

Lemma 8. *Let $M = C(M_1, \ldots, M_n)$, $N = D(N_1, \ldots, N_p)$, with C and D two distinct constructors. Let $\mathcal{E}$ be a set of equations satisfying (1) and such that $M =_\mathcal{E} N$. Then $\mathcal{E}$ does not satisfy (2).*

Proof. Let σ be any substitution substituting ground terms for every variable occurring in M or N. From $M =_\mathcal{E} N$ we get

$$\sigma(M) = C(\sigma(M_1), \ldots, \sigma(M_n)) =_\mathcal{E} D(\sigma(N_1), \ldots, \sigma(N_p)) = \sigma(N)$$

and therefore by (1)

$$C(\mathcal{GC}_\mathcal{E}[\sigma(M_1)], \ldots, \mathcal{GC}_\mathcal{E}[\sigma(M_n)]) =_\mathcal{E} D(\mathcal{GC}_\mathcal{E}[\sigma(N_1)], \ldots, \mathcal{GC}_\mathcal{E}[\sigma(N_p)])$$

a contradiction with (2). ∎

Lemma 9. *Let $M = C(M_1, \ldots, M_n)$, with C in C, and let N be a variable. Let $\mathcal{E}$ be a set of equations satisfying (1) and such that $M =_\mathcal{E} N$. Then $\mathcal{E}$ does not satisfy (2).*

Proof. Let σ be any substitution that replaces N by a term whose leading function symbol is a constructor distinct from C (Remember that we assume the existence of at least two constructors.) We have $\sigma(M) =_\mathcal{E} \sigma(N)$, and the result follows from the preceding lemma ∎

We are now ready to present our extension of the Knuth-Bendix completion algorithm.

4. The Inductive Completion Algorithm

Let $\mathcal{E}$ satisfying the principle of definition, $\mathcal{E}'$ any set of Σ-equations. Run the Knuth-Bendix completion

algorithm on $\mathcal{E} \cup \mathcal{E}'$, with the following modifications. We assume given a well-founded partial ordering on terms $\succ$, compatible with the term structure and stable by substitution, with which we prove the termination of the successive sets of rewrite rules. We assume familiarity with the Knuth-Bendix completion algorithm, as presented for instance in Huet[10]. The only modification occurs in the step in which a pair of terms, coming from either a simplified critical pair or a reduced rewrite rule, is considered for orientation before being added as a new rewrite rule. This step should be modified as follows, assuming that (M, N) is the current candidate rewrite rule, with $M \neq N$.

- *If* $M = C(M_1, \ldots, M_n)$ *with* C in $\mathcal{C}$, *then:*
  - *If* $N = C(N_1, \ldots, N_n)$ *then* replace the pair by the n pairs (M_i, N_i)
  - *If* $N = D(N_1, \ldots, N_p)$, *with* D *in* $\mathcal{C}$, $D \neq C$, *or* $N = x$ *stop* "disproof"
  - *Otherwise:*
    - *If* $N \succ M$, introduce new rule $N \rightarrow M$
    - *Otherwise stop* "failure"
- *Otherwise:*
  - *If* $N = C(N_1, \ldots, N_n)$ *with* C *in* $\mathcal{C}$ do symetrically as above
  - *Otherwise:*
    - *If* $M \succ N$, introduce new rule $M \rightarrow N$
    - *If* $N \succ M$, introduce new rule $N \rightarrow M$
    - *Otherwise stop* "failure". ∎

The new inductive completion algorithm may:

• stop with success, i.e. we get a finite canonical term rewriting system.

• stop with disproof.

• stop with failure, i.e. either the ordering $\succ$ used was inadequate to prove the termination of the set of current rewrite rules, or this set is nonterminating, and the method is therefore not applicable.

• run forever, generating an infinite set of rewrite rules.

Theorem. *If the algorithm stops with success, every equation in* $\mathcal{E}'$ *holds in* $I(\Sigma, \mathcal{E})$. *Furthermore, the resulting canonical term rewriting system satisfies the principle of definition. If the algorithm stops with disproof, some equation in* $\mathcal{E}'$ *does not hold in* $I(\Sigma, \mathcal{E})$. *Conversely, if some equation in* $\mathcal{E}'$ *does not hold in* $I(\Sigma, \mathcal{E})$, *the algorithm stops with either disproof or failure.*

Proof. From the lemmas above and the properties of the Knuth-Bendix algorithm, as proved in Huet[10]. ∎

Note that lemma 5 forbids us to introduce a left-hand side whose main function symbol is a constructor. This is necessary, as shown by the following example. Let $C = \{A, B\}$, $D = \{C\}$, all symbols nullary. Let $\mathcal{E} = \{C = A\}$, and $\mathcal{E}' = \{C = B\}$. With $A \succ C$ and $B \succ C$, the usual completion algorithm would converge with the canonical set $\{A \rightarrow C, B \rightarrow C\}$. The algorithm above would oblige us to use an ordering such that $C \succ A$ and $C \succ B$, would construct for $\mathcal{E} \cup \mathcal{E}'$ the term rewriting system $\{C \rightarrow A, C \rightarrow B\}$, and the critical pair (A, B) would (correctly) force stopping with disproof.

The theorem above was inspired by the work of Musser[15], and its extensions in the taut presentations of Goguen[6] and the s-separable theories of Huet and Oppen[11]. However, note that here no special equality axioms are required.

5. General Organization of Inductive Proofs

Assume we are working in the initial theory $I(\Sigma, \mathcal{E})$, with $\Sigma = \mathcal{C} \uplus \mathcal{D}$. That is, we are interested in studying properties of the objects freely constructed from members of $\mathcal{C}$, and to this end we have axiomatized the operators of $\mathcal{D}$ using the equations in $\mathcal{E}$ as recursive definitions.

We check that every left-hand side of $\mathcal{E}$ is of the form $F(M_1, \ldots, M_n)$ with $F \in \mathcal{D}$, that $\mathcal{E}$ is confluent, nœtherian, and verifies the hypothesis of lemma 4. These checks usually go together: if $\mathcal{E}$ is a set of primitive recursive-like definitions, it can be proved nœtherian by a simple lexicographic ordering argument, every defined function symbol has trivially a complete set of arguments, and the set is confluent because there are no critical pairs.

Now let $\mathcal{E}'$ be a set of equations which we conjecture about the inductive theory above. We run the inductive completion algorithm above, initializing the set of rewrite rules to $\mathcal{E}$ and the set of equations to $\mathcal{E}'$. If the algorithm stops with failure, nothing interesting may be said. If it stops with disproof, some equation from $\mathcal{E}'$ does not hold in the theory. If it stops with success, generating a canonical set $\mathcal{E}_1$, all of the equations from $\mathcal{E}'$ are true in $I(\Sigma, \mathcal{E}) = I(\Sigma, \mathcal{E}_1)$, and furthermore $\mathcal{E}_1$ satisfies the definition principle. We may therefore iterate the process, trying a new set of conjectures $\mathcal{E}'_1$, while profiting of the previously proved conjectures as lemmas.

Let us now consider the situation when we want to enrich our theory with new function symbols. First of all, we remark that it would be unsound to add new constructors, since a theory complete for $\mathcal{C}$ might not be complete for some extended $\mathcal{C}'$. Furthermore

we may have proved some theorem valid in $I(\Sigma, \mathcal{E})$ which is *not* valid in the extended theory $I(\Sigma', \mathcal{E}')$. For instance, with $C = \{A, B\}$ and $\mathcal{E} = \{F(A) = A, F(B) = B\}$ we may prove $F(x) = x$, but this formula is not valid any more if we extend our theory with constructor C and definition $F(C) = A$. We shall therefore assume that our set of constructors C is constant, and that we only permit to enrich our signature by adding new defined function symbols. We are sure this way that our extensions are monotonous, in the sense that any theorem proved in a theory is still true in an extended theory, even if we do not keep it around as a lemma.

Assume therefore that we are currently dealing with a set of equations $\mathcal{E}$ that is known to satisfy the definition principle, and that we are adding a new function symbol F and some new definitions $\mathcal{E}'$. How do we know that $\mathcal{E} \cup \mathcal{E}'$ satisfies the definition principle? Our problem is that $\mathcal{E}$ itself may not satisfy the hypothesis of lemma 4, because $\mathcal{E}$ may be obtained after some steps of completion that may have destroyed the completeness property. For instance, consider $C = \{\mathbf{true}, \mathbf{false}\}$, $D = \{\vee\}$, $\mathcal{E} = \{\mathbf{true} \vee x = \mathbf{true}, x \vee \mathbf{true} = \mathbf{true}, \mathbf{false} \vee \mathbf{false} = \mathbf{false}\}$. If we attempt to prove the theorem $x \vee x = x$ using the completion algorithm, we shall stop with success, generating the canonical set $\mathcal{E}' = \{\mathbf{true} \vee x = \mathbf{true}, x \vee \mathbf{true} = \mathbf{true}, x \vee x = x\}$. This set is known to satisfy the definition principle, but lemma 4 does not apply to it any more, and therefore cannot be used to show that some extension of it satisfies the definition principle for an extended signature. However, the following slight generalization of lemmas 4 and 5 will be enough to tell us how to enrich canonical theories while preserving the definition principle.

Assume that $\mathcal{E}$ obeys the hypothesis of lemma 5 and is known to have property (1) relatively to a given signature $\Sigma = C \uplus D$. Now assume we want to enrich our theory by one more defined symbol F, i.e. we now consider signature $\Sigma' = C \uplus (D \cup \{F\})$. Consider any set $\mathcal{E}'$ obtained from $\mathcal{E}$ by adding a complete definition for F, i.e. a set of equations with left-hand sides of the form $F(S_i^1, \ldots, S_i^n)$, the argument tuples S_i's forming a complete set. If $\mathcal{E}'$ is canonical, it satisfies the principle of definition for the extended signature. This way we know how to test the validity of our definition principle in an incremental way. Actually, remark that the process of enriching a theory, once the completeness property has been checked, is exactly the same as proving lemmas: it just consists in running the completion algorithm. This is probably the most surprising feature of our theorem-prover: we treat new axioms and conjectures to be proven in exactly the same manner.

In practice it will be useful to deal with sorted theories. Over sorted theories, we shall be able to introduce new constructors, provided we introduce at a time all constructors of a given sort, and that none of the symbols considered so far had arguments of the new sort. For instance, we may consider introducing sort `boolean` with constructors **true** and **false**, define certain boolean connectives and prove properties about them, then introduce sort `integer` with constructors 0 and S, prove arithmetic properties, then introduce `list-of-integers` with constructors *Null* and *List*, etc...

The Appendix presents examples of proofs and disproofs using the method above. All our examples satisfy the hypotheses of lemmas 4 and 5 above, as the reader may readily check. However, in the current implementation these conditions are not checked automatically. We plan to implement fully the method above, using for the termination tests recent criteria developed by Plaisted[17], Dershowitz[4], and Kamin & Lévy[12].

6. Extension to Commutative-Associative Operators

The theory above can be extended without difficulty to the generalization of the Knuth-Bendix completion algorithm to the case where certain function symbols are assumed to be commutative and associative [14,16]. These operators must be placed in D. For these operators, the notion of a complete set of tuples of arguments extends naturally to the notion of a complete set of multisets of arguments.

In the Appendix we apply this technique to proofs of simple arithmetic identities. In particular, we show that the sum of the first n odd integers is n^2, and that the sum of the first n integers is $\frac{n(n+1)}{2}$.

It appears possible too to introduce commutative-associative contructors. This would allow proofs of recursive programs over data types such as multisets. However, lemma 7 must be changed accordingly; that is, an equation $C(M_1, \ldots, M_n) = C(N_1, \ldots, N_p)$, with $C \in C$ and C commutative-associative, does not necessarily imply pairwise equality of the arguments M_i and N_i. It rather implies one out of the possibly several solutions to the corresponding multiset equation. This would complicate the general organization of the method, since the proofs would have to split according to the various cases. We have not yet implemented this mechanism in our proof system.

Conclusion

We have presented in this paper a very simple method to construct proofs by structural induction. The method is based on a straightforward modification of the Knuth-Bendix completion algorithm, and does not require an equality axiomatization. The method is simple to implement, and when it applies the proofs obtained are surprisingly short. For instance, given the two recursive definitions of the concatenation of lists, we can prove the associativity of concatenation by simply checking that this set of three equations, considered as rewrite rules, forms a canonical set.

Experimental evidence with an implementation of our method suggests that it is powerful enough to apply to the usual proofs of correctness of algebraic data types implementation, and proofs of simple primitive recursive programs computing on data types such as integers, lists and trees. We know how to handle simple fragments of arithmetic, and thus generate automatically proofs of standard summation identities.

The method has many limitations however. The requirement on finite termination, while natural for recursive definitions (or recursive programs, provided we restrict ourselves to programs that always terminate), may be impossible to enforce for complex combinations of lemmas. We do not know how to handle permutative equations, except for commutative-associative laws. Even when we know how to give an orientation to all the generated equations so that finite termination holds, the completion process may loop. It may however be possible to recognize easy patterns of such loopings, and avoid these by appropriate "meta-rules", such as the generalization technique of Boyer and Moore. Finally, most nontrivial program proofs involve a fair amount of propositional calculus (such as cases analysis). Such reasoning is better dealt with as a separate top-level proof system rather than by equational encoding.

Appendix

The following is the image of a computer session run on SRI's KL10 using the VLISP interpreter developed at Université de Vincennes. User input appears after question marks. When in doubt, the system asks the user the orientation of equation $M = N$ with the prompt COMMAND ? to which one answers y (resp. n) to get the rewrite rule $M \rightarrow N$ (resp. $N \rightarrow M$). Comments are inclosed between semi-colons.

```
;We start with a simple axiomatization of list
structures;
?  (initialization)
   List of constructors ?  (NULL CONS)
   List of AC operators ?  ()
   List of infix operators ?  ()
   List of data-files ?  (lisp)
   Mode (Free/Const/Auto) ?  const
;we enter the definitions of append and reverse;
?  (complete app&rev)

------------------------------------------------------------

   Given set of equations: APP&REV

APPEND(NULL,x) = x
APPEND(CONS(x,y),z) = CONS(x,APPEND(y,z))
REV(NULL) = NULL
REV(CONS(x,y)) = APPEND(REV(y),CONS(x,NULL))

------------------------------------------------------------

R1 : APPEND(NULL,x) → x                                  Given

------------------------------------------------------------

R2 : APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
                                                         Given

------------------------------------------------------------

R3 : REV(NULL) → NULL                                    Given

------------------------------------------------------------

REV(CONS(x,y)) = APPEND(REV(y),CONS(x,NULL))  Given

   Command ? y

R4 : REV(CONS(x,y)) → APPEND(REV(y),CONS(x,NULL))

------------------------------------------------------------

   Complete Set: *APP&REV

   Unification time: 113ms
   Rewriting  time: 280ms

;we now prove rev(rev(x))=x;
?  (prove rev.rev)

------------------------------------------------------------

  Given set of equations: REV.REV

REV(REV(x)) = x

------------------------------------------------------------

R5 : REV(REV(x)) → x                                     Given

------------------------------------------------------------

R6 : REV(APPEND(REV(x),CONS(y,NULL))) → CONS(y,x)
                                          From R5 and R4

------------------------------------------------------------

R7 : REV(APPEND(x,CONS(y,NULL))) → CONS(y,REV(x))
                                          From R6 and R5

R6 deleted
Rewrite rules :   R7 R5  for left part

------------------------------------------------------------

   Complete Set: *REV.REV

   Unification time: 410ms
   Rewriting  time: 1003ms

?  (show *rev.rev)

------------------------------------------------------------
```

```
*REV.REV

APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
REV(NULL) → NULL
REV(CONS(x,y)) → APPEND(REV(y),CONS(x,NULL))
REV(REV(x)) → x
REV(APPEND(x,CONS(y,NULL))) → CONS(y,REV(x))
```

```
 ;Let us now consider a new function brev.
In pseudo-LISP notation, we would program:
brev(x)=if null(x) then nil
        elsif null(cdr(x)) then list(x)
        else cons(car(brev(cdr(x))),
     brev(cons(car(x),brev(cdr(brev(cdr(x)))))))).
We shall here define brev with the help of auxiliary
functions brev1 and brev2, such that
        brev1(x,y)=car(brev(cons(x,y))) and
        brev2(x,y)=cdr(brev(cons(x,y))).
Note that our "programs" are closer to Burstall's
HOPE than to LISP;
?  (complete *rev.rev brev)
```

```
   Given set of equations: BREV

BREV(NULL) = NULL
BREV(CONS(x,y)) = CONS(BREV1(x,y),BREV2(x,y))
BREV1(x,NULL) = x
BREV1(x,CONS(y,z)) = BREV1(y,z)
BREV2(x,NULL) = NULL
BREV2(x,CONS(y,z)) = BREV(CONS(x,BREV(BREV2(y,z))))
```

```
R7 : BREV(NULL) → NULL                                  Given
```

```
R8 : BREV(CONS(x,y)) → CONS(BREV1(x,y),BREV2(x,y))
                                                        Given
```

```
R9 : BREV1(x,NULL) → x                                  Given
```

```
BREV1(x,CONS(y,z)) = BREV1(y,z)                         Given
   Command ? y

R10 : BREV1(x,CONS(y,z)) → BREV1(y,z)
```

```
R11 : BREV2(x,NULL) → NULL                              Given
```

```
R12 : BREV2(x,CONS(y,z)) → CONS(BREV1(x,BREV(
   BREV2(y,z))),BREV2(x,BREV(BREV2(y,z))))   Given
```

```
   Complete Set: *BREV

   Unification time: 557ms
   Rewriting   time: 695ms

?  (show *brev)
```

```
*BREV

APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
REV(NULL) → NULL
REV(CONS(x,y)) → APPEND(REV(y),CONS(x,NULL))
REV(REV(x)) → x
REV(APPEND(x,CONS(y,NULL))) → CONS(y,REV(x))
BREV(NULL) → NULL
BREV(CONS(x,y)) → CONS(BREV1(x,y),BREV2(x,y))
BREV1(x,NULL) → x
BREV1(x,CONS(y,z)) → BREV1(y,z)
BREV2(x,NULL) → NULL
BREV2(x,CONS(y,z)) → CONS(BREV1(x,BREV(BREV2(y,
                    z))),BREV2(x,BREV(BREV2(y,z))))
```

```
;brev is actually still another reverse function,
as we now show;
?  (prove rev.brev)
```

```
  Given set of equations: REV.BREV

BREV(x) =REV(x)
```

```
BREV(x) = REV(x)                                        Given
   Command ? y

R13 : BREV(x) → REV(x)

R7 deleted
Rewrite rules :  R13 R3  for left part

R8 replaced by:
APPEND(REV(x),CONS(y,NULL)) = CONS(BREV1(y,x),
                                   BREV2(y,x))
Rewrite rules:   R13 R4  for left part

R12 replaced by:
BREV2(x,CONS(y,z)) = CONS(BREV1(x,REV(BREV2(y,
                    z))),BREV2(x,REV(BREV2(y,z))))
Rewrite rules:  R13  for right part
```

```
R14 : BREV2(x,CONS(y,z)) → CONS(BREV1(x,REV(BREV2(y,
   z))),BREV2(x,REV(BREV2(y,z))))          From R12
```

```
R15 : APPEND(REV(x),CONS(y,NULL)) → CONS(BREV1(y,
   x),BREV2(y,x))                          From R8

R4 replaced by:
REV(CONS(x,y)) = CONS(BREV1(x,y),BREV2(x,y))
Rewrite rules:   R15  for right part
```

```
R16 : REV(CONS(x,y)) → CONS(BREV1(x,y),BREV2(x,y))
                                          From R4
```

```
R17 : BREV1(BREV1(x,y),BREV2(x,y)) → x
                                  From R16 and R5
```

```
R18 : BREV2(BREV1(x,y),BREV2(x,y)) → y
                                  From R16 and R5
```

```
R19 : APPEND(x,CONS(y,NULL)) → CONS(BREV1(y,REV(x)),
   BREV2(y,REV(x)))                    From R15 and R5

R6 deleted
Rewrite rules :   R19 R16 R17 R18  for left part

R15 deleted
Rewrite rules :   R19 R5 R5  for left part
```

```
   Complete Set: *REV.BREV

   Unification time: 1684ms
   Rewriting   time: 6433ms
```

```
  *REV.BREV

APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
REV(NULL) → NULL
REV(REV(x)) → x
BREV1(x,NULL) → x
BREV1(x,CONS(y,z)) → BREV1(y,z)
BREV2(x,NULL) → NULL
BREV(x) → REV(x)
BREV2(x,CONS(y,z)) → CONS(BREV1(x,REV(BREV2(y,z)))
                          ,BREV2(x,REV(BREV2(y,z))))
REV(CONS(x,y)) → CONS(BREV1(x,y),BREV2(x,y))
BREV1(BREV1(x,y),BREV2(x,y)) → x
BREV2(BREV1(x,y),BREV2(x,y)) → y
APPEND(x,CONS(y,NULL)) → CONS(BREV1(y,REV(x)),
                               BREV2(y,REV(x)))
```

```
;note the use of our induction rule in the proof
above, in generating R17 and R18.
Let us now play with "distributive cons";
?  (complete dcons)
```

```
   Given set of equations: DCONS

DCONS(x,NULL) =NULL
DCONS(x,CONS(y,z)) =CONS(CONS(x,y),DCONS(x,z))
```

```
R1 : DCONS(x,NULL) → NULL                          Given
```

```
R2 : DCONS(x,CONS(y,z)) → CONS(CONS(x,y),DCONS(x,z))
                                                   Given
```

```
   Complete Set: *DCONS

   Unification time: 60ms
   Rewriting   time: 19ms

;we now enter the function iterate, and prove a
lemma relating iterate and dcons;
?  (prove iterate)
```

```
  Given set of equations: ITERATE

ITERATE(NULL,x) =NULL
ITERATE(CONS(x,y),z) =CONS(z,ITERATE(y,z))
DCONS(x,ITERATE(y,z)) =ITERATE(y,CONS(x,z))
```

```
R3 : ITERATE(NULL,x) → NULL                        Given
```

```
R4 : ITERATE(CONS(x,y),z) → CONS(z,ITERATE(y,z))
                                                   Given
```

```
DCONS(x,ITERATE(y,z)) = ITERATE(y,CONS(x,z))   Given
   Command ? y
R5 : DCONS(x,ITERATE(y,z)) → ITERATE(y,CONS(x,z))
```

```
   Complete Set: *ITERATE

   Unification time: 155ms
   Rewriting   time: 395ms

;we now enter the definition of vm, a function
that repeats a vector as a matrix;
?  (complete *iterate vm)
```

```
   Given set of equations: VM

VM(NULL) = NULL
VM(CONS(x,y)) = CONS(CONS(x,y),DCONS(x,VM(y)))
```

```
R6 : VM(NULL) → NULL                               Given
```

```
R7 : VM(CONS(x,y)) → CONS(CONS(x,y),DCONS(x,VM(y)))
                                                   Given
```

```
   Complete Set: *VM

   Unification time: 99ms
   Rewriting   time: 73ms

?  (show *vm)
```

```
  *VM

DCONS(x,NULL) → NULL
DCONS(x,CONS(y,z)) → CONS(CONS(x,y),DCONS(x,z))
ITERATE(NULL,x) → NULL
ITERATE(CONS(x,y),z) → CONS(z,ITERATE(y,z))
DCONS(x,ITERATE(y,z)) → ITERATE(y,CONS(x,z))
VM(NULL) → NULL
VM(CONS(x,y)) → CONS(CONS(x,y),DCONS(x,VM(y)))
```

```
;we now express vm in terms of iterate;
?  (prove vm.iterate)
```

```
  Given set of equations: VM.ITERATE

VM(x) =ITERATE(x,x)
```

```
VM(x) = ITERATE(x,x)                               Given
   Command ? y

R8 : VM(x) → ITERATE(x,x)


R6 deleted
Rewrite rules :   R8 R3  for left part

R7 deleted
Rewrite rules :   R8 R4  for left part
                  R8 R5  for right part
```

```
   Complete Set: *VM.ITERATE

   Unification time: 147ms
   Rewriting   time: 273ms

;we now enter a new function itvm, together with
append;
?  (complete *vm.iterate itvm)
```

```
   Given set of equations: ITVM

ITVM(NULL,x,y) = y
ITVM(CONS(x,y),z,u) = ITVM(y,z,CONS(z,u))
APPEND(NULL,x) = x
APPEND(CONS(x,y),z) = CONS(x,APPEND(y,z))
```

```
R7 : ITVM(NULL,x,y) → y                                    Given
```

```
ITVM(CONS(x,y),z,u) = ITVM(y,z,CONS(z,u))                  Given
   Command ? y

R8 : ITVM(CONS(x,y),z,u) → ITVM(y,z,CONS(z,u))
```

```
R9 : APPEND(NULL,x) → x                                    Given
```

```
R10 : APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
                                                           Given
```

```
   Complete Set: *ITVM

   Unification time: 378ms
   Rewriting   time: 532ms
?  (show *itvm)
```

```
  *ITVM

DCONS(x,NULL) → NULL
CONS(x,CONS(y,z)) → CONS(CONS(x,y),DCONS(x,z))
ITERATE(NULL,x) → NULL
ITERATE(CONS(x,y),z) → CONS(z,ITERATE(y,z))
DCONS(x,ITERATE(y,z)) → ITERATE(y,CONS(x,z))
VM(x) → ITERATE(x,x)
ITVM(NULL,x,y) → y
ITVM(CONS(x,y),z,u) → ITVM(y,z,CONS(z,u))
APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
```

```
;we now express itvm in terms of iterate;
?  (prove itvm.iterate)
```

```
  Given set of equations: ITVM.ITERATE

ITVM(x,y,z) =APPEND(ITERATE(x,y),z)
```

```
ITVM(x,y,z) = APPEND(ITERATE(x,y),z)                       Given
   Command ? y

R11 : ITVM(x,y,z) → APPEND(ITERATE(x,y),z)

R7 deleted
Rewrite rules :   R11 R3 R9  for left part
R8 replaced by:
CONS(x,APPEND(ITERATE(y,x),z)) =
              APPEND(ITERATE(y,x),CONS(x,z))
Rewrite rules:    R11 R4 R10  for left part
                  R11  for right part
```

```
R12 : APPEND(ITERATE(x,y),CONS(y,z)) →
      CONS(y,APPEND(ITERATE(x,y),z))                       From R8
```

```
   Complete Set: *ITVM.ITERATE

   Unification time: 324ms
   Rewriting   time: 591ms

;and finally, we show that we may compute vm
iteratively, since:       vm(x)=itvm(x,x,null).
This proof is done by mere simplification, once we
prove the lemma:          append(x,null)=x;
?  (prove vm.itvm)
```

```
  Given set of equations: VM.ITVM

APPEND(x,NULL) = x
VM(x) = ITVM(x,x,NULL)
```

```
R11 : APPEND(x,NULL) → x                                   Given
```

```
   Complete Set: *VM.ITVM

   Unification time: 151ms
   Rewriting   time: 205ms

?  (show *vm.itvm)
```

```
  *VM.ITVM

DCONS(x,NULL) → NULL
DCONS(x,CONS(y,z)) → CONS(CONS(x,y),DCONS(x,z))
ITERATE(NULL,x) → NULL
ITERATE(CONS(x,y),z) → CONS(z,ITERATE(y,z))
DCONS(x,ITERATE(y,z)) → ITERATE(y,CONS(x,z))
VM(x) → ITERATE(x,x)
APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
ITVM(x,y,z) → APPEND(ITERATE(x,y),z)
APPEND(ITERATE(x,y),CONS(y,z)) →
              CONS(y,APPEND(ITERATE(x,y),z))
APPEND(x,NULL) → x
```

```
;We thank Patrick Greussay for suggesting the
example above.
Let us now give an example of disproof;
?  (show *rev)
```

```
  *REV

APPEND(NULL,x) → x
APPEND(CONS(x,y),z) → CONS(x,APPEND(y,z))
REV(NULL) → NULL
REV(CONS(x,y)) → APPEND(REV(y),CONS(x,NULL))
```

```
? (complete *rev wrong)
```

```
   Given set of equations: WRONG

REV(x) = APPEND(x,x)
```

```
REV(x) = APPEND(x,x)                                    Given
   Command ? y

R5 : REV(x) → APPEND(x,x)

R3 deleted
Rewrite rules :   R5 R1  for left part

R4 replaced by:
CONS(x,APPEND(y,CONS(x,y))) =
                  APPEND(APPEND(y,y),CONS(x,NULL))
Rewrite rules:    R5 R2  for left part
                  R5  for right part
```

```
R6 : APPEND(APPEND(x,x),CONS(y,NULL)) →
     CONS(y,APPEND(x,CONS(y,x)))                  From R4
```

```
NULL = CONS(x,NULL)                          From R6 and R1
**************************************************
*****************     FAILED     *******************
**************************************************

;We now give examples of the method with
commutative-associative operators;
? (initialization)

   List of constructors  ?  (0 S)
   List of AC operators  ?  (+ *)
   List of infix operators  ?  (+ * ↑)
   List of data-files    ?  (arith)
   Inside-Out Reductions    ?  y
? (complete sum.of.odds)
```

```
   Given set of equations: SUM.OF.ODDS

1 = S(0)
2 = S(1)
n+0 = n
n+S(m) = S(n+m)
n*0 = 0
n*S(m) = (n*m)+n
n↑0 = 1
n↑S(m) = n*(n↑m)
SIGMA(0) = 0
SIGMA(S(n)) = SIGMA(n)+((2*n)+1)
```

```
R1 : 1 → S(0)                                           Given
```

```
R2 : 2 → S(S(0))                                        Given
```

```
R3 : n+0 → n                                            Given
```

```
R4 : n+S(m) → S(n+m)                                    Given
```

```
R5 : n*0 → 0                                            Given
```

```
n*S(m) = (n*m)+n                                        Given

   Command ? y

R6 : n*S(m) → (n*m)+n
```

```
R7 : n↑0 → S(0)                                         Given
```

```
n↑S(m) = n*(n↑m)                                        Given

   Command ? y

R8 : n↑S(m) → n*(n↑m)
```

```
R9 : SIGMA(0) → 0                                       Given
```

```
R10 : SIGMA(S(n)) → S(SIGMA(n)+n+n)                     Given
```

```
   Complete Set: *SUM.OF.ODDS

   Unification time: 977ms
   Rewriting   time: 3116ms

? (prove identity)
```

```
  Given set of equations: IDENTITY

SIGMA(n) = n↑2
```

```
SIGMA(n) = n*n                                          Given

   Command ? y

R11 : SIGMA(n) → n*n

R9 deleted
Rewrite rules :   R11 R5  for left part

R10 deleted
Rewrite rules :   R11 R6 R4 R6  for left part
                  R11  for right part
```

```
   Complete Set: *IDENTITY

   Unification time: 50ms
   Rewriting   time: 742ms

? (show *identity)
```

```
  *IDENTITY

1 → S(0)
2 → S(S(0))
n+0 → n
n+S(m) → S(n+m)
n*0 → 0
n*S(m) → (n*m)+n
n↑0 → S(0)
n↑S(m) → n*(n↑m)
SIGMA(n) → n*n
```

```
; We are now interested in showing that
            SUM(n)=n*(n+1)/2,
where SUM(n) denotes the sum of the first n integers:
            SUM(n)=1+2+...+n.
To achieve this aim, we introduce the HALF function.
Then we prove the following lemma:
            HALF(2n+m)=n+HALF(m).
And finally we prove the summation identity:
            SUM(n)=HALF(n*(n+1)).
We start with the following complete set:
R1 :   n+0 → n
R2 :   n+S(m) → S(n+m)
R3 :   n*0 → 0
R4 :   n*S(m) → (n*m)+n
R5 :   HALF(0) → 0
R6 :   HALF(S(0)) → 0
R7 :   HALF(S(S(n))) → S(HALF(n))
We first prove the lemma concerning HALF;
?  (prove lemma.HALF)
```

```
  Given set of equations: LEMMA.HALF

HALF(n+n+m) =n+HALF(m)
```

```
HALF(n+n+m) = n+HALF(m)                          Given

   Command ? y

R8 : HALF(n+n+m) → n+HALF(m)
```

```
R9 : HALF(n+n) → n                          From R8 and R1
```

```
HALF(S(n+n+m)) = n+HALF(S(m))               From R8 and R2

   Command ? y
R10 : HALF(S(n+n+m)) → n+HALF(S(m))
```

```
R11 : HALF(S(n+n)) → n                      From R10 and R1
```

```
   Complete Set: *LEMMA.HALF

   Unification time: 2130ms
   Rewriting   time: 12942ms
; R7, R9 and R11 are the usual properties of HALF,
R8 and R10 being generalizations of these properties.
Let us now introduce the definition of SUM;
?  (complete *lemma.half def.sum)
```

```
  Given set of equations: DEF.SUM

SUM(0) =0
SUM(S(n)) =SUM(n)+S(n)
```

```
R12 : SUM(0) → 0                                 Given
```

```
R13 : SUM(S(n)) → S(SUM(n)+n)                    Given
```

```
   Complete Set: *DEF.SUM

   Unification time: 199ms
   Rewriting   time: 579ms

;We are now ready to prove the summation identity;
?  (prove proof.sum)
```

```
  Given set of equations: PROOF.SUM

SUM(n) =HALF(n*S(n))
```

```
SUM(n) = HALF((n*n)+n)                           Given
   Command ? y

R14 : SUM(n) → HALF((n*n)+n)

R12 deleted
Rewrite rules : R14 R3 R1 R5  for left part

R13 deleted
Rewrite rules : R14 R4 R2 R4 R2 R2 R8 R7  for left
part
                R14  for right part
```

```
   Complete Set: *PROOF.SUM

   Unification time: 49ms
   Rewriting   time: 1523ms

?  (Show *PROOF.SUM)
```

```
  *PROOF.SUM

n+0 → n
n+S(m) → S(n+m)
n*0 → 0
n*S(m) → (n*m)+n
HALF(0) → 0
HALF(S(0)) → 0
HALF(S(S(n))) → S(HALF(n))
HALF(n+n+m) → n+HALF(m)
HALF(n+n) → n
HALF(S(n+n+m)) → n+HALF(S(m))
HALF(S(n+n)) → n
SUM(n) → HALF((n*n)+n)
```

Acknowlegments

This research was carried out during a visit to the Computer Science Laboratory of SRI International. We had numerous discussions on the topic of inductive proofs with Bob Boyer, Joe Goguen, Patrick Greussay, J Moore and Rob Shostak. We are very grateful to Jérôme Chailloux for transporting VLISP on the KL10 at SRI. We thank Stanford University for allowing us to typeset this paper in TEX on the SAIL computer.

References

1. Aubin R., *Mechanizing Structural Induction*. Ph.D. thesis, U. of Edinburgh, Edinburgh 1976.
2. Boyer R. and Moore J, *Proving Theorems About LISP Functions*. JACM 22 (1975), 129–144.
3. Boyer R. and Moore J, *A Computational Logic*. Academic Press, 1979.
4. Burstall R.M., *Proving Properties of Programs by Structural Induction*. Computer J. 12 (1969), 41–48.
5. Dershowitz N., *Orderings for Term-rewriting Systems*. Proc. 20th Symposium on Foundations of Computer Science (1979), 123–131. To appear, Theoretical Computer Science.
6. Goguen J.A., *How to Prove Algebraic Inductive Hypotheses Without Induction, With Applications to the Correctness of Data Type Implementation*. Proceedings of the Fifth Conference on Automated Deduction, Les Arcs, July 1980.
7. Goguen J.A., Thatcher J.W., Wagner E.G. and Wright J.B., *Initial Algebras Semantics and Continuous Algebras*. JACM 24 (1977), 68–95.
8. Goguen J.A., Thatcher J.W. and Wagner E.G., *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*. "Current Trends in Programming Methodology", Vol 4, Ed. Yeh R., Prentice-Hall (1978), 80–149.
9. Huet G., *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*. 18th IEEE Symposium on Foundations of Computer Science (1977), 30–45. To appear, JACM, October 1980.
10. Huet G., *A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm*. Unpublished manuscript, March 1980. Rapport de Recherche INRIA to appear.
11. Huet G. and Oppen D., *Equations and Rewrite Rules: a Survey*. "Formal Languages: Perspectives and Open Problems". Ed. Book R., Academic Press, 1980. Also Technical Report CSL-111, SRI International, January 1980.
12. Kamin S. and Lévy J.J., *Private communication*. February 1980.
13. Knuth D. and Bendix P., *Simple Word Problems in Universal Algebras*. "Computational Problems in Abstract Algebra". Ed. Leech J., Pergamon Press, 1970, 263–297.
14. Lankford D.S. and Ballantyne A.M., *Decision Procedures for Simple Equational Theories With Commutative-Associative Axioms: Complete Sets of Commutative-Associative Reductions*. Report ATP 39, Departments of Mathematics and Computer Sciences, U. of Texas at Austin, Aug. 1977.
15. Musser D. L., *On Proving Inductive Properties of Abstract Data Types*. Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Jan. 1980, 154–162.
16. Peterson G.E. and Stickel M.E., *Complete Sets of Reductions for Equational Theories With Complete Unification Algorithms*. Tech. Report, Dept. of Computer Science, U. of Arizona, Tucson, Sept. 1977.
17. Plaisted D., *A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems*. Dept. of Computer Science Report 78–943, U. of Illinois at Urbana-Champaign, Sept. 1978.