Proof by Consistency*

Deepak Kapur and David R. Musser

Computer Science Branch, Corporate Research and Development, General Electric Company, Schenectady, NY 12345, U.S.A.

Recommended by Robert Boyer and Alan Bundy

ABSTRACT

Advances of the past decade in methods and computer programs for showing consistency of proof systems based on first-order equations have made it feasible, in some settings, to use proof by consistency as an alternative to conventional rules of inference. Musser described the method applied to proof of properties of inductively defined objects. Refinements of this inductionless induction method were discussed by Kapur, Goguen, Huet and Hullot, Huet and Oppen, Lankford, Dershowitz, Paul, and more recently by Jouannaud and Kounalis as well as by Kapur, Narendran and Zhang. This paper gives a very general account of proof by consistency and inductionless induction, and shows how previous results can be derived simply from the general theory. New results include a theorem giving characterizations of an unambiguity property that is key to applicability of proof by consistency, and a theorem similar to the Birkhoff’s Completeness Theorem for equational proof systems, but concerning inductive proof.

1. Introduction

A familiar method of mathematical proof is:

Proof by contradiction. Assume the negation of the formula that is to be proved, and show there is a contradiction.

Much less familiar is:

Proof by consistency. Assume the formula that is to be proved, and show there is no contradiction.

The reader may be surprised to see the second approach called a proof
method, because it would seem to be unsound, allowing the proof of formulas which are not valid. This is of course correct: the second approach cannot be used in general; it can be applied, however, when one has a \textit{strongly complete} proof system. By definition, a proof system $S$ is strongly complete if every formula $P$ of $S$ that is consistent with $S$ is a theorem of $S$ (see, for example, Hilbert and Ackermann [8] or Hughes and Creswell [13, pp.19–21]). By $P$ is \textit{consistent with} $S$ we mean that the proof system obtained from $S$ by adding $P$ as an axiom is consistent. Another way to express strong completeness is to say that there are no formulas that can be added as independent axioms. It is easy to show that strong completeness implies the more commonly encountered notion of (weak) completeness, that every valid formula is a theorem of the system.

The usual formulations of propositional calculus are consistent and strongly complete, and those of first-order predicate calculus are consistent and weakly complete but not strongly complete (since, for example, the formula $\forall x P(x)$ can be added consistently although it is not a theorem). Another example is the Peano axiomatization of natural number arithmetic, which is not even weakly complete by Godel's Incompleteness Theorem. However, when restricted to formulas that are universally quantified equations, the Peano system is strongly complete.

It might seem obvious why proof by consistency has rarely been used in practice: the apparent difficulty of proving consistency of formal systems. However, over the past decade or so a great deal of progress has been made in developing methods of \textit{dynamically generating decision procedures}. We are thinking in particular of the progress in the area of equational decision procedures generated by the Knuth-Bendix [19] approach, which makes it quite feasible to carry out the needed consistency proofs in many cases. This progress makes it worth seriously considering whether proof by consistency might now be a feasible approach.

Using proof by consistency, one would prove a given equational formula of number theory, for example, $P = \forall n Q(n)$, where

$$Q(n) = \left[ 2 \sum_{i=1}^{n} i = n(n + 1) \right]$$

by assuming it as an axiom along with the Peano axioms (without the induction axiom schema) and showing the consistency of the resulting system. (An interested reader may consult Appendix A where an automatic proof of this property done on RRL, a rewrite rule laboratory, under development at General Electric Corporate Research and Development and Rensselaer Polytechnic Institute, is given.) This method of proof is in stark contrast to the usual proof by mathematical induction, i.e., the direct application of the Peano's induction schema.
As a simple example of the way proof by consistency works, suppose we define $+$ on natural numbers by:

\[
\begin{align*}
    n + 0 &= n, \quad (1) \\
    n + \text{suc}(m) &= \text{suc}(n + m), \quad (2) \\
    \text{suc}(m) + n &= \text{suc}(m + n), \quad (3)
\end{align*}
\]

where $\text{suc}$ is the successor function, and a function $\text{double}$ by:

\[
\begin{align*}
    \text{double}(0) &= 0, \quad (4) \\
    \text{double}(\text{suc}(n)) &= \text{suc}(\text{suc}(\text{double}(n))). \quad (5)
\end{align*}
\]

We then attempt to prove

\[
\text{double}(n) = n + n. \quad (6)
\]

This can be done using the Peano induction schema:

- **Basis case:** $\text{double}(0) = 0 + 0$.
- **Induction step:** $\text{double}(n) = n + n$ implies $\text{double}(\text{suc}(n)) = \text{suc}(n) + \text{suc}(n)$.

Both of these cases follow from (1)–(5).

Using proof by consistency, one adds (6) to (1)–(5) and attempts to show the resulting equations are consistent. To do so, one can use the Knuth-Bendix procedure to generate a rewrite rule decision procedure for (1)–(6). The details of the theoretical basis for and computations involved in the Knuth-Bendix procedure are discussed elsewhere; the main calculations are of superpositions of left-hand sides of equations and resulting \textit{critical pairs} of terms which are then reduced as far as possible using the equations as rewrite rules. Rules corresponding to (1)–(5) themselves form a decision procedure for the equational theory of $\text{suc}$, $+$, and $\text{double}$. When the rule corresponding to (6) is added, (4) and (6) have a superposition corresponding to the basis case:

\[
\begin{array}{c}
\text{double}(0) \\
(4) \\
0 \\
\end{array}
\quad \begin{array}{c}
\text{double}(0) \\
(6) \\
0 + 0
\end{array}
\]

\begin{align*}
\text{double}(0) &= 0 \\
\text{double}(\text{suc}(n)) &= \text{suc}(\text{suc}(\text{double}(n)))
\end{align*}

\[
\rightarrow
\begin{align*}
\text{double}(0) &= 0 \\
\text{double}(\text{suc}(n)) &= \text{suc}(\text{suc}(\text{double}(n)))
\end{align*}
\]
Similarly, (5) and (6) have a superposition corresponding to the induction step:

```
        double(suc(n))
         /     \
       (5)     (6)
  suc(suc(double(n)))  suc(n) + suc(n)
           \      /  \(2)
            suc(suc(n) + n)
             /  \(3)
      suc(suc(n + n))
```

In each case the critical pair of terms are reduced to the same term. One can also show that any sequence of reductions using (1)–(6) as rewrite rules always terminates. Together these facts show that the rules constitute a decision procedure that can decide when two terms are equal as an equational consequence of (1)–(6). This decision procedure can in turn be used to check for consistency, basically because it can answer the question of whether any two values that are distinct are equateable using (1)–(6); e.g., whether 1 = 0 is a consequence of (1)–(6). (These statements are made precise in later sections of the paper.)

It is worth noting that the critical pair computations are very similar to the calculations involved in the traditional induction schema proof; the work is just organized a bit differently. The advantage of proof by consistency is that the organization of the proof can often be handled more automatically than with the induction schema proof, which in general requires several nontrivial steps: (1) choice of induction variable; (2) applications of the appropriate equations to make equational substitutions, and (3) renaming bound variables and choosing an appropriate instantiation in applying the induction hypothesis.

The use of proofs of consistency, in lieu of induction, was first proposed by Musser [22] and has since been refined by Kapur [16], Goguen [6], Huet and Hullot [12], Huet and Oppen [11], Lankford [20, 21], Dershowitz [4], Paul [24], and more recently by Jouanaud and Kounalis [14] and Kapur, Narendran and Zhang [18]. Lankford dubbed the method inductionless induction. The context of most of this work has been efforts to automate (at least partially) the proof of theorems in systems based on the Peano axioms or analogous axiomatizations of other domains of interest in computation, such as finite sequences, strings, queues, arrays, etc. Most work on automating mathematical induction proofs (we mention also the important work of Boyer and

---

1The defining equations (1)–(5) must also be shown to be consistent. Equation (3) has some superpositions with other equations but the critical pair analysis shows it to be consistent with the other equations. Actually, this shows that (3) is itself an inductive consequence of (1) and (2). The role of (3) in the above proof by consistency will be discussed further later in the paper.
Moore [2]) has been motivated at least partially by program verification research.

In this paper, we study the general theory of proof by consistency and inductionless induction. Unlike earlier works on inductionless induction which have been done in the context of algebraic specification of abstract data types, the results in this paper are developed in a general algebraic framework. In Sections 3–9, the methods of proof by consistency and inductionless induction are related to inductive models and inductive completeness, strong completeness, and initial and final algebras. Perhaps the most interesting of the new results presented are:

- a theorem giving characterizations of an unambiguity property that is key to applicability of proof by consistency, and
- a theorem similar to Birkhoff's Completeness Theorem for equational proof, but concerning inductive proof.

Section 10 contains a number of examples of applications to abstract data type specifications. In Section 11, the limitation of the commonly used definition of sufficient completeness of an abstract data type specification is discussed and a new definition is proposed based on these investigations. Section 12 contains a sketch of an extension of proof by consistency to ambiguous systems.

A brief survey of previous work on inductionless induction is given in Section 2.

2. Related Work

Musser [22] was the first to show that for data type specifications satisfying certain restrictions, proof by induction of an equation \(e_1 = e_2\) from a specification \(S\) was the same as showing consistency of \(S \cup \{e_1 = e_2\}\). The restriction on each specification was what Musser called the full-specification property: that every data type have an equality predicate as an operation which is completely specified on the values of the data type in the sense that for any two specific values, whether the equality predicate is true or false could be deduced in the equational theory. Musser also demonstrated how this method of proving inductive properties could be implemented by using the Knuth–Bendix completion procedure to show the consistency of an equational specification.

Kapur [16] introduced the notion of a distinguishability theory, \(DS(S)\), of a specification \(S\) of a data type; \(DS(S)\) is an extension of the equational theory, \(EQ(S)\), of \(S\) and includes inequalities of the form \(e_1 \neq e_2\) along with equations. In addition to the rules of equality, there is an extra rule of inference in \(DS(S)\):

**Proof by contradiction.** \(e_1 \neq e_2\) in \(DS(S)\) if it is possible to derive a contradiction in the equational theory of \(S \cup \{e_1 = e_2\}\).

Deriving a contradiction is the same as showing inconsistency of \(S \cup \{e_1 = e_2\}\).
The inequality "true ≠ false" is assumed to serve as the basis for deducing inequalities. A system $S$ is called contradictory (or inconsistent) if and only if it is possible to deduce (i) $true = false$ in $EQ(S)$ or (ii) $e_3 = e_4$ in $EQ(S)$ for some $e_3$ and $e_4$, and at the same time, $e_3 \neq e_4$ is in $DS(S)$. In [16], the completeness property of a specification $S$ is defined as: for every pair $c_1, c_2$ of constants such that $\text{typeof}(c_1) = \text{typeof}(c_2)$ either $c_1 = c_2$ is in $DS(S)$ or $c_1 \neq c_2$ is in $DS(S)$. Intuitively, this notion of completeness means that from the specification it is possible to deduce the relation of a constant to every other constant. Musser's full-specification property implies completeness. Under certain conditions, the inverse also holds, i.e., completeness implies full specification. It is observed that for a complete specification $S$, proof by induction is the same as showing the consistency of an extended theory.

Goguen's treatment [6] of proof by induction and his definition of $s$-taut for every type $s$ in the specification $S$ is related to Kapur's treatment [16]. Instead of defining a distinguishability theory of $S$, Goguen requires that for every constant pair $c_1, c_2$, if $c_1 = c_2$ is not in $EQ(S)$, then there must be a boolean-valued expression $u(x)$ with a single variable $x$, such that $\text{equal}(u(c_1), u(c_2)) = false$ in $EQ(S)$; this requirement implies that $c_1 \neq c_2$ is in $DS(S)$, as in $EQ(S \cup \{c_1 = c_2\})$, we can derive $true = false$. Huet and Oppen [11] give a treatment similar to that of Musser [22]; instead of including the equality predicate as an operation of a data type, they require that the distinguishability predicate (negation of equality) be included as an operation of a data type and be totally specified.

Huet and Hullot [12] refined Musser's method and incorporated it into the Knuth–Bendix completion procedure for data types whose values can be generated in a unique way using an identifiable set of constructors. They require that every operation other than the constructors (called a derived function) satisfy the principle of definition, which requires that for every constant $c$ in $G$ (built using derived operations and constructors), there is a unique $c'$ in $GC$, the set of constants generated using the constructors, such that $c = c'$ in $EQ(S)$; this condition guarantees that the axioms completely define the derived operations. They used a meta-axiom scheme inside the Knuth–Bendix completion procedure; this meta-axiom scheme asserts that (syntactically) different constants built using only constructors are not equal to each other, i.e., for $c_1$ and $c_2$ in their $GC$, if $c_1$ and $c_2$ are not the same, then $c_1 \neq c_2$ is in $DS(S)$. Using this meta-axiom scheme and the principle of definition on every derived function, Huet and Hullot's method ensures the full-specification property of data types. To prove an equation $e_1 = e_2$ by induction from $S$, the Knuth–Bendix completion procedure is applied on $S \cup \{e_1 = e_2\}$; if the completion procedure stops without generating any equation which relates distinct terms built using constructors, then $e_1 = e_2$ is in the inductive theory of $S$; if the completion procedure generates an equation which relates distinct terms built using constructors (which means that the extended $S$ is inconsistent), then $e_1 = e_2$ is not in the inductive theory of $S$. Huet and
Hullot also discuss how to extend the results in case of associative/commutative axioms.

Lankford [20, 21] stated conditions similar to those of Huet and Hullot for derived functions to be completely defined and discussed this method in the framework of term rewriting systems by relating confluence to consistency. Lankford also discusses extensions of this approach to algebras and data types which have associative/commutative operations.

The approach of Dershowitz [4] is related to Huet and Hullot’s work and Lankford’s ideas. If a specification $S$ can be transformed to a canonical set of rewrite rules $R$ such that only the desired constants serve as the canonical forms, the completeness of $S$ is guaranteed; for every pair of constants $c_1, c_2$, either $\text{canon}(c_1) = \text{canon}(c_2)$ or $\text{canon}(c_1) \neq \text{canon}(c_2)$, where $\text{canon}(c)$ is the irreducible constant corresponding to $c$ under $R$. Now from $S$, $t = u$ is deducible inductively if $S \cup \{t = u\}$ does not make two distinct canonical forms of $S$ equal. If the consistency property of $S$ is viewed as that all irreducible constants under $R$ be distinct, then the last statement is equivalent to requiring that the consistency of $S \cup \{t = u\}$ be preserved.

Some of the preliminary ideas of the present paper first germinated in Musser and Kapur [23]. We later discovered that it is sufficient to require that the data types be sufficiently complete in order to do proofs by induction by reducing them to consistency. This result was based on the fact that the final model [15, 25] of a sufficiently complete specification was unique and results of Musser, Kapur, Goguen, Huet and Hullot, and Lankford explicitly or indirectly depended on this property. We subsequently developed a generalization of these results in a model-theoretic (algebraic) framework without having to depend upon any specific properties of abstract data types; the rest of the paper discusses these results.

After a preliminary version of this paper appeared, a paper by Paul [24] was brought to our attention. Results reported in that paper are similar to the results in Lankford [20, 21] and Dershowitz [4]. Some concepts in [24] are related to the concepts discussed in this paper, though his definition of an inductively complete specification is different from the definition of an inductively complete relation introduced in this paper.

More recently, Jouannaud and Kounalis [14] as well as Kapur, Narendran, and Zhang [18] proposed methods for proving inductive properties using the Knuth–Bendix’s completion procedure in the presence of relations over constructors. These methods can be considered as different ways to implement the general approach of proof by consistency for unambiguous systems as elaborated in this paper.

3. Basic Definitions

Let $L$ be a language of terms generated from a given finite set $F$ of function symbols and a given denumerably infinite set $V$ of variable symbols, disjoint
from $F$. A term in $L$ is either a variable from $V$ or consists of a function symbol from $F$ and a finite sequence of terms (called arguments); we use the usual denotations with parentheses and commas; e.g., $f(g(\ ), v)$ denotes a term with two argument terms, $g(\ )$ and the variable $v$, where $g(\ )$ is a constant term, i.e., having zero arguments. When it is clear from the context that $g$ is a function symbol and not a variable, such terms will be written without the "()", e.g., $f(g, v)$. (In a later section we will restrict the language to terms obeying type restrictions, but for this section those restrictions are unnecessary.)

Terms containing no variables are called ground terms. In this and the next two sections, for simplicity, we permit only ground terms, but we will extend the definitions and theory developed here to general terms in Section 6.

An equation is a pair of terms, $(t, u)$, usually written $t = u$. Given a set $E$ of equations between ground terms, we denote by $=^E$ the smallest congruence relation containing $E$.

Define a system $S$ as $(L, C, E)$, where $L$ is a language of terms, $C$ is a subset of the ground terms of $L$ containing at least two elements, and $E$ is a set of equations between ground terms of $L$. The purpose of $C$ is to designate a set of terms that cannot be equated to each other. When $C$ is large or infinite, then it can be specified by a scheme such as by giving the function symbols that generate the constants in $C$. In this paper, we do not get into the details of how $C$ can be specified. One common way is to designate a subset of functions as constructors and make $C$ to be a subset of ground terms built from constructors (called constructor ground terms) that has at most one representative from every equivalence class induced by $E$ over constructor ground terms (see for example [12, 14, 18]).

We say that $M$ is a model of $S = (L, C, E)$ if $M$ is a model of $E$ and every pair of distinct ground terms $c, d$ in $C$ have distinct interpretations in $M$.

(As usual) we say that $S$ is consistent if it has a model. The following theorem gives a characterization of consistency that will be used frequently below.

**Theorem 3.1.** A system $S$ is consistent if and only if there is no pair of distinct ground terms $c, d$ in $C$ such that $c =^E d$.

**Remark.** For algebraic specifications of data types, it is usual to specify a boolean data type in which $true$ and $false$ are constant terms; then $C$ would contain (at least) $true$ and $false$, forbidding an equation $true = false$.

**Proof.** ($\Rightarrow$): Consistency of $S$ implies there is a model $M$ in which $c, d$ have distinct interpretations. However $M$ must also be a model of $E$, which means that $c =^E d$ would imply that $c$ and $d$ have the same interpretation.

($\Leftarrow$): Take the quotient algebra on ground terms defined by $L$ with
partition induced by the congruence relation $=_{E'}$. It can easily be verified that this is a model of $S$. \qed

We now define a relation $\sim$, read *possibly equal to*, on the ground terms of $L$ by:

$$t \sim u \text{ if and only if for } E' = E \cup \{t = u\}, \text{ the system } S' = (L, C, E') \text{ is consistent.}$$

By the preceding theorem, $t \sim u$ if and only if there is no pair of distinct ground terms $c, d$ in $C$ such that $c =_{E'} d$. The negation $\neq$ of $\sim$ is essentially the *distinguishability* relation of Kapur [16].

The relation $\sim$ is empty if $S$ itself is inconsistent. However, if $S$ is consistent, the following lemma shows that $\sim$ is *almost* a congruence relation, lacking, in general, only the property of transitivity.

**Lemma 3.2.** If $S$ is consistent, then $\sim$ is reflexive, symmetric, and has the substitution property.

**Proof.** Reflexivity and symmetry follow directly from the definition. To prove the substitution property, let $t \sim u$ and let $f$ be any function symbol of $L$; we have to show that $f(\ldots, t, \ldots) \sim f(\ldots, u, \ldots)$. Let

$$E' = E \cup \{t = u\}$$

and

$$E'' = E \cup \{f(\ldots, t, \ldots) = f(\ldots, u, \ldots)\}.$$ 

Then $=_{E''}$ is contained in $=_{E'}$, hence any relation between ground terms $c, d$ of $C$ obtainable with $=_{E'}$ could also be obtained with $=_{E''}$, but by $t \sim u$ there is no such relation; hence $S'' = (L, C, E'')$ is consistent. \qed

**Example 3.3.** Let

$$S_1 = ((a, b, c, d), \{c, d\}, \{\}).$$

The $=_{E}$ relation is the identity relation, $\{(a), \{b\}, \{c\}, \{d\}\}$. The relation $\sim$ is

$$\{(a, a), (a, b), (a, c), (a, d), (b, a), (b, b), (b, c), (b, d), (c, a), (c, b), (c, c), (d, a), (d, b), (d, d)\}.$$

The relation $\sim$ is not a congruence relation as it is not transitive. These relations can be pictorially expressed as in Fig. 1.
Example 3.4. Let

\[ S_2 = (\{a, b, c, d\}, \{c, d\}, \{a = b\}) \]

The \(=_E\) relation is \(\{\{c\}, \{a, b\}, \{d\}\}\). The \(\sim\) relation is the same as for the \(\sim\) relation for \(S_1\). A pictorial representation of these relations is shown in Fig. 2.

We now give a system \(S_3\) such that its relation \(\sim\) is a congruence relation.

Example 3.5. Let

\[ S_3 = (\{a, b, c, d\}, \{c, d\}, \{a = b, a = c\}) \]

The \(=_E\) relation is \(\{\{c, a, b\}, \{d\}\}\). The relation \(\sim\) is the same as \(=_E\). Figure 3 is the picture corresponding to this example.

4. \(S\)-congruence relations

Although \(\sim\) is not quite a congruence relation, it is closely related to an important class of congruence relations:
An *S-congruence relation* is any congruence relation that contains \( \approx_E \) and respects \( C \), i.e., it contains no congruence between distinct elements of \( C \).

**Example 4.1.** For the system \( S \) discussed above, the following relations are \( S \)-congruence relations.

1. \( \{\{a, b, c\}, \{d\}\} \).
2. \( \{\{c\}, \{a, b, d\}\} \).
3. \( \{\{a, c\}, \{b, d\}\} \).
4. \( \{\{b, c\}, \{a, d\}\} \).
5. \( \{\{a, b\}, \{c\}, \{d\}\} \).

These relations can be pictorially represented as in Fig. 4. For the system \( S_2 \), the relations (1), (2), and (5) are \( S_2 \)-congruence relations.

![Fig. 4. Pictorial representation of the \( S_1 \)-congruence relations (1)–(5) (Example 4.1).](image)

**Lemma 4.2.** Any \( S \)-congruence relation is contained in \( \sim \).

**Proof.** Let \( \equiv_1 \) be any \( S \)-congruence relation and suppose \( t \equiv_1 u \). We have to show that \( t \sim u \). Suppose not, then there are distinct \( c, d \) in \( C \) such that \( c \equiv_E d \) where \( E' = E \cup \{t = u\} \). Let \( c = c_0, c_1, \ldots, c_n = d \) and suppose there is a use of \( t = u \) in \( c \equiv_1 c_{i+1} \); then also \( c_i \equiv_1 c_{i+1} \) since \( i \equiv_1 u \) and \( \equiv_1 \) is a congruence relation. For \( c_i = c_{i+1} \) in which an equation of \( E \) is used, we again have \( c_i \equiv_1 c_{i+1} \) since any \( S \)-congruence relation contains \( \equiv_1 \). Thus, \( c = c_0, \ldots, c_n = d \) and \( c \equiv_1 d \), but this contradicts the assumption that \( \equiv_1 \) is an \( S \)-congruence relation and must therefore respect \( C \). \( \square \)

An \( S \)-congruence relation \( \equiv_1 \) is *maximal* if there is no pair \( (t, u) \) of ground terms for which the congruence closure of \( \equiv_1 \cup \{(t, u)\} \) is an \( S \)-congruence relation. Maximal \( S \)-congruence relations can be constructed by running through an enumeration of all pairs of ground terms, for each pair adding it when possible (and taking the congruence closure). For examples, the \( S_1 \)-congruence relations (1)–(4) in Example 4.1. for the system \( S_1 \) are maximal, whereas the fifth \( S_1 \)-congruence relation (5) is not maximal.

The notion of a maximal \( S \)-congruence relation is related to maximal consistent extensions and to final models. Kamin [15] discussed a related approach to specifying abstract data types using final models.
Lemma 4.3. If \( t \sim u \), then there is a maximal \( S \)-congruence relation \( =_1 \) such that \( t =_1 u \).

Proof. Take \( =_1 \) as any maximal congruence relation contained in \( \sim \) and containing \( =_E \cup \{ (t, u) \} \). \( \square \)

Combining the previous two lemmas, we obtain

Theorem 4.4. The relation \( \sim \) is the union of all of the maximal \( S \)-congruence relations.

5. The Inductive Congruence Relation

While \( \sim \) has been shown to be the union of all of the maximal \( S \)-congruence relations, we define the inductive congruence relation, denoted by \( =_I \), as the intersection of all maximal \( S \)-congruence relations. The name inductive will be justified by theorems proved below. We also pronounce \( t =_I u \) as \( t \) is necessarily equal to \( u \).

Theorem 5.1. A system \( S \) is consistent if and only if a nonempty \( =_I \) exists.

Proof. \( (\Rightarrow) \): A maximal \( S \)-congruence relation can be generated by enumerating all pairs that are not in \( =_E \) and trying to add them to \( =_E \) without destroying consistency. This ensures that there is at least one maximal \( S \)-congruence relation. Since \( =_E \) is contained in every \( S \)-congruence relation, the intersection of all maximal \( S \)-congruence relations is not empty.

\( (\Leftarrow) \): If \( =_I \) exists, then \( S \) is consistent as \( =_I \), and hence \( =_E \), respects \( C \). \( \square \)

The following lemma shows that the inductive congruence relation is a congruence relation.

Lemma 5.2. The inductive congruence relation is an \( S \)-congruence relation.

Proof. The relation \( =_I \) contains \( =_E \) and is contained in \( \sim \) because it is the intersection of relations with those properties. Furthermore, any relation defined as the intersection of congruence relations is itself a congruence relation. \( \square \)

The relation \( =_I \) is not necessarily maximal among \( S \)-congruence relations. For example, the first four \( S_1 \)-congruence relations listed above for the system \( S_1 \) are the only maximal \( S_1 \)-congruence relations, while the \( =_I \) for \( S_1 \) is the identity relation. However, for \( S_2 \), the \( =_I \) relation is \( \{ (a, b), (c), (d) \} \), the same as \( =_E \).
The following theorem answers the question of when $\equiv$ is maximal, and gives other important properties of the relationship between $\equiv$ and $\sim$.

**Theorem 5.3.** Let $S$ be consistent (hence $\sim$ is nonempty). Then the following statements are equivalent:

(a) $\sim$ is an $S$-congruence relation;
(b) there is a unique maximal $S$-congruence relation (and it is $\sim$);
(c) $\equiv$ is the same relation as $\sim$;
(d) $\sim$ is transitive.

**Proof.** (a) $\Rightarrow$ (b): Since every $S$-congruence relation is contained in $\sim$ (Lemma 4.2), any $S$-congruence distinct from $\sim$ would have to omit some pair $(t, u)$ such that $t \sim u$, and thus could not be maximal. Thus, $\sim$ is the unique maximal $S$-congruence relation.

(b) $\Rightarrow$ (c): Follows from the definition of $\equiv$.

(c) $\Rightarrow$ (d): Any congruence relation is transitive.

(d) $\Rightarrow$ (a): By Lemma 3.2 and transitivity, $\sim$ is a congruence relation. It is an $S$-congruence relation since it contains $\equiv$ and respects $C$. $\square$

One may think that for a consistent ground system, i.e., in which $E$ involves only ground term, $\equiv = \equiv_{E}$. That is not the case, as the following example illustrates.

**Example 5.4.** Consider a system $\text{unary} = (L, C, E)$, where $L$ includes a nullary function $a$ and a unary function $f$, $C = \{a, f(a)\}$, and $E = \{f(f(f(a))) = a\}$. It is easy to see that $\langle f(f(a)), a \rangle$ is not in $\equiv_{E}$, whereas $\langle f(f(a)), a \rangle$ is in $\equiv$. It can be verified that there is only one maximal $S$-congruence relation for $\text{unary}$.

We define a system $S$ to be *ambiguous* if it has more than one maximal $S$-congruence relation, and to be *unambiguous* if it has a unique maximal $S$-congruence relation. Thus, the preceding theorem gives several characterizations of unambiguity of a system $S$.

For example, systems $S_1$ and $S_2$ are ambiguous, whereas $S_3$ is unambiguous. The system $\text{unary}$ of Example 5.4 is also unambiguous. As another example, consider the following.

**Example 5.5.** Consider a system $\text{Bool} = (L, C, E)$, where $L$ includes nullary functions $\text{true}$, $\text{false}$, and the unary function $\text{not}$. Let $C = \{\text{true}, \text{false}\}$ and $E$ be the set

$\{\text{not(true)} = \text{false}, \text{not(false)} = \text{true}\}$.

Then $\text{Bool}$ is unambiguous, but if either of the equations in $E$ were omitted, the resulting system would be ambiguous.
6. Extension to General Terms

Up to this point we have been considering only relations between ground terms. We now consider relations between general terms, i.e., terms (possibly) containing variables. At this point it is useful to introduce types (or sorts) and make our languages of terms many-sorted (although there may be some applications in which the following would be useful even for only one sort).

By a prior definition, a language \( L \) is the set of terms generated by a given finite set \( F \) of function symbols and a given denumerably infinite set \( V \) of variable symbols, disjoint from \( F \). Let \( W \) be a set of symbols called type names, disjoint from \( F \) and \( V \). We now assume that each function symbol \( f \) in \( F \) has associated with it a sequence \( (a_1, \ldots, a_n, r) \) of type names from \( W \), where \( n \geq 0; a_1, \ldots, a_n \) are the argument types for \( f \) and \( r \) is the range type. This sequence is called the arity of \( f \) and is conventionally denoted as \( f : a_1 \times \cdots \times a_n \rightarrow r \).

For example, the arities of the functions of the \( \text{Bool} \) system introduced in Example 5.5 are (using \( \text{Bool} \) also as a type name):

\[
\begin{align*}
true & : \rightarrow \text{Bool}, \\
false & : \rightarrow \text{Bool}, \\
not & : \text{Bool} \rightarrow \text{Bool}.
\end{align*}
\]

We might also introduce another type name, \( \text{Nat} \), and another function symbol and arity, \( \text{binary} : \text{Bool} \rightarrow \text{Nat} \). (We could then have equations such as \( \text{binary}(true) = 1, \text{binary}(false) = 0 \).)

We further assume that each type name, \( T \), has associated with it a certain finite set of function symbols, called the functions belonging to \( T \), including at least one function whose range type is \( T \). We say that type \( T \) directly depends on the functions appearing in the arities of the functions belonging to \( T \), other than \( T \) itself. For example, if \( \text{binary} \) belongs to \( \text{Bool} \) as well as \( \text{true}, \text{false} \), and \( \text{not} \), then \( \text{Bool} \) directly depends on \( \text{Nat} \).

For \( T \) and \( U \) in the reflexive, transitive closure of the directly-depends relation, we say that \( T \) depends on \( U \). For example, \( \text{Bool} \) depends on \( \text{Bool} \) and \( \text{Nat} \), and on any type that \( \text{Nat} \) depends on.

Now, for each type name, \( T \), we will define the language of \( T \), denoted \( L(T) \). We give an inductive definition, which at the same time defines a mapping \( \text{TypeOf} \) from terms to type names.

(a) For each type \( U \) on which \( T \) depends, an infinite set of variable symbols \( v \) with \( \text{TypeOf}(v) = U \), is contained in \( L(T) \).

(b) For each function symbol \( f \) belonging to a type on which \( T \) depends, let \( a_1 \times \cdots \times a_n \rightarrow r \) be its arity, where \( n \geq 0 \); then for each tuple \( (e_1, \ldots, e_n) \) of terms \( e_i \) in \( L(T) \) with \( \text{TypeOf}(e_i) = a_i \), for \( i = 1, \ldots, n \), the term \( f(e_1, \ldots, e_n) \) is in \( L(T) \) with \( \text{TypeOf}(f(e_1, \ldots, e_n)) = r \).
(c) No other terms are in $L(T)$.

Note that these definitions permit mutual dependencies between types, as the induction is on the structure of terms rather than on any hierarchy in the dependency relation. For example, $Nat$ might well depend on $Bool$, in addition to $Bool$ depending on $Nat$ as above.

We can now define the language $L$ of a system to be union of the languages $L(T)$ for each of the type names $T$ in the set of type names $W$. (We could have defined a typed language without the subdivision into languages $L(T)$, but the subdivision is essential for later definitions (Section 11).)

In the following, for any binary relation on terms such as $\sim$, $=_{E'}$, $=_{E}$, etc., we require that $TypeOf(t) = TypeOf(u)$ for $\langle t, u \rangle$ to be in the relation.

We also require that substitutions obey types; i.e., $TypeOf(\sigma(v)) = TypeOf(v)$ for any variable $v$. This implies that for any substitution instance $\sigma(t)$ of $t$, $TypeOf(\sigma(t)) = TypeOf(t)$.

We now extend the definitions of $=_{E}$, $\sim$, $S$-congruence relations, and $=$ to apply to all terms of $L(T)$, with the convention that whenever a pair $\langle t, u \rangle$ belongs to one of these relations, so do all substitution instances of that pair. This gives $=_{E}$ its usual meaning, and the reader can check that because of the way the other relations are defined in terms of $=_{E}$, they are well-defined. (Note that we do not apply this convention to the complements of these relations.)

Furthermore, each of the lemmas and theorems of the previous section still holds with these extended definitions, as the reader may also verify.

### 7. Induction

A relation is said to be inductively complete if and only if for every pair of terms $\langle t, u \rangle$, if for every ground substitution $\sigma$, the pair $\langle \sigma(t), \sigma(u) \rangle$ is in the relation, then the pair $\langle t, u \rangle$ is in the relation also.

The relation $\sim$ is not necessarily inductively complete; neither is the relation $=_{E}$. For an example in which $\sim$ is not inductively complete, consider

**Example 7.1.**

$$S_4 = (\{f, g, a, b, c, d\}, \{c, d\}, \{f(g(a), g(b)) = c, f(a, b) = d\}),$$

where $f$ is binary, $g$ is unary, and $a, b, c, \text{ and } d$ are constant symbols. It can be shown that $\langle g(a), a \rangle, \langle g(b), b \rangle, \langle g(c), c \rangle, \text{ and } \langle g(d), d \rangle$ are in the $\sim$ relation of $S_4$; however, $\langle g(x), x \rangle$ is not in $\sim$. This is so because $S_4'$ in which $E' = \{f(g(a), g(b)) = c, f(a, b) = d, g(x) = x\}$ is inconsistent since $c =_{E} d$ using both $g(a) = a$ and $g(b) = b$. 


As an example of \( =_E \) not being inductively complete, let us consider again the system \( \text{Bool} \) and consider terms (both ground as well as nonground).

**Example 7.2.**

\[
\text{Bool} = (\{\text{not, true, false}\}, \{\text{true, false}\}, \\
\{\text{not(true) = false, not(false) = true}\}).
\]

Here \( =_E \) is not inductively complete because the pair \( \langle \text{not(not(x)), x} \rangle \) is not in \( =_E \), even though for every ground substitution \( \sigma \), \( \langle \sigma(\text{not(not(x))), \sigma(x) \rangle \) is in \( =_E \). However, \( \text{Bool}'s \sim \) and \( = \) relations, which are the same, are inductively complete. Figure 5 depicts this example.

![Fig. 5. Pictorial representation of Example 7.2.](image)

**Example 7.3.** Consider a modification of \( \text{Bool} \) in which \( E \) does not include the equation: \( \text{not(false) = true} \); let us call the modified system \( \text{Bool}' \). For \( \text{Bool}' \), the relation \( \sim \) includes \( \langle \text{not(false), false} \rangle \) as well as \( \langle \text{not(false), true} \rangle \); \( \sim \) also includes \( \langle \text{not(not(x)), x} \rangle \) as it is consistent with \( E' \). Figure 6 depicts this modification. The reader can contrast Fig. 6 with Fig. 5.

![Fig. 6. Pictorial representation of Example 7.3.](image)

Although \( \sim \) and \( =_E \) are not necessarily inductively complete, we have:

**Theorem 7.4.** Every maximal \( S \)-congruence relation is inductively complete.

**Proof.** Let \( =_1 \) be any maximal \( S \)-congruence relation. Let \( \langle t, u \rangle \) be any pair of terms such that for all ground substitutions \( \sigma \), \( \sigma(t) =_1 \sigma(u) \). We have to show that \( t =_1 u \). Suppose that, on the contrary, \( t =_1 u \). Let \( =_2 \) be the congruence closure of \( =_1 \cup \langle t, u \rangle \). Since \( =_1 \) is maximal, \( =_2 \) must not respect \( C \), hence
there are distinct $c, d$ in $C$ such that $c =_{2} d$. Furthermore (by Birkhoff's Completeness Theorem—see Section 8), there are terms $c_{1}, \ldots, c_{n}$ such that

$$c = c_{1} =_{2} \cdots =_{2} c_{n} = d.$$ 

By our assumption that all substitution instances of any related pair of terms are also in the relation, we can assume that the $c_{i}$ are all ground terms. But then each $=_{2}$ in this chain can be replaced by $=_{1}$, since each link is either due to $=_{1}$ or to a ground instance of $<t, u>$, and we have assumed that all ground instances of $<t, u>$ are in the $=_{1}$ relation. Thus, we have $c =_{1} d$, a contradiction since $=_{1}$ respects $C$. □

From this theorem we easily obtain the following key result, which justifies the name inductive given to $=_{1}$:

**Theorem 7.5.** The $=_{1}$ relation is inductively complete.

**Proof.** Follows directly from the definition of $=_{1}$, Theorem 7.4, and the fact that the intersection of inductively complete relations is inductively complete. □

**Corollary 7.6.** In any system, $t =_{1} u$ if and only if for every ground substitution $\sigma$, $\sigma(t) =_{1} \sigma(u)$.

8. A Birkhoff-like Theorem of Inductive Completeness

Birkhoff [1] showed the completeness of the rules of inference of equality:

**Theorem 8.1 (Birkhoff’s Completeness Theorem).** $t =_{E} u$ if and only if $t = u$ is true in every model of $E$.

Recall that $t =_{E} u$ means proof from $E$ using strictly equational rules of inference: reflexivity, symmetry, transitivity, and substitutivity. Recall also that being a model of $E$ is weaker than being a model of a system $S$, as there isn’t any chosen set of ground terms $C$ such that in each model distinct elements of $C$ have distinct interpretations.

We define an inductive model of a system $S$ to be a model $M$ of $S$ such that no proper epimorphic image of $M$ is a model of $S$.

The following theorem is a completeness result analogous to Birkhoff’s Theorem, but for $=_{1}$ instead of for $=_{E}$.

**Theorem 8.2 (Inductive Completeness Theorem).** $t =_{1} u$ if and only if $t = u$ is true in every inductive model of $S$. 
Remark. In terms of proof theory, we regard $\equiv$ as meaning use of the equational rules of inference plus the following inductive rule of inference:

$$\begin{align*}
\text{for every ground substitution } \sigma, \sigma(t) &\equiv \sigma(u) \\
T &\equiv u
\end{align*}$$

Such a rule of inference cannot be used directly in proofs because it has infinitely many premises, but it can be replaced by a rule having only finitely many premises corresponding to the structure of the ground terms—structural induction, or, more generally, induction based on any well-founded ordering of the ground terms.

Since the inductive congruence relation is defined as the intersection of all maximal $S$-congruence relations, the proof of the Inductive Completeness Theorem reduces to proving the following:

Lemma 8.3. A model of $S$ is inductive if and only if it is isomorphic to a model induced by some maximal $S$-congruence relation.

Proof. ($\Leftarrow$): Let $=_{1}$ be any maximal $S$-congruence relation. Define $M$ to be the quotient algebra defined by $L$ with partition induced by the congruence relation $=_{1}$. It can easily be verified that $M$ is a model of $S$. Let $M_{1}$ be any proper epimorphic image $M_{1}$ of $M$. Then there are distinct $=_{1}$ congruence classes in $M$, represented by some terms $t$ and $u$ such that in $M_{1}$, $t$ and $u$ have the same interpretation. But $t =_{1} u$ and $=_{1}$ is maximal, so adding $(t, u)$ to $=_{1}$ would be inconsistent; hence $t$ and $u$ having the same interpretation in $M_{1}$ means $M_{1}$ does not respect $C$ and thus cannot be a model of $S$.

($\Rightarrow$): Let $M$ be an inductive model of $S$. Construct a congruence relation $=_{1}$ on $L$ as follows: Initially define $=_{1}$ to be the empty relation. Let $(t_{i}, u_{i}), i = 1, 2, \ldots$, be an exhaustive enumeration of pairs of terms of $L$, and consider each pair; it is included in $=_{1}$ if and only if $t_{i}$ and $u_{i}$ have the same interpretation in $M$. Then $=_{1}$ is an $S$-congruence, for it contains $=_{1}$ by Birkhoff's Completeness Theorem, and it respects $C$ since $M$ is an $S$ model. Suppose it is not maximal. Then there is some $(t, u)$ that could be added to $=_{1}$ without causing inconsistency, but this pair was left out because $t$ and $u$ have different interpretations in $M$. Hence we can define another model $M_{1}$ of $S$ by mapping the elements of $M$ onto themselves, except that $t$ is mapped onto $u$. This is an epimorphism to $M_{1}$, hence $M$ is not an inductive model, contrary to assumption. 

9. Strong Completeness and Inductive Completeness of Unambiguous Systems

We define a relation on a system $S$ to be strongly complete if no pair of terms can be added to the relation without making $S$ inconsistent. Note that $\sim$ is by
definition strongly complete, and since = and ~ are the same relation in an unambiguous system (Theorem 5.3), we have:

**Lemma 9.1.** In an unambiguous system = is strongly complete.

Combining this with Corollary 7.6, we obtain the desired connection between strong completeness and inductive completeness that justifies proof (of inductive properties) by consistency.

**Theorem 9.2.** In an unambiguous system \( S = (L, C, E) \), the following are equivalent:

(a) \( t = u \),

(b) for every ground substitution \( \sigma \), \( \sigma(t) = \sigma(u) \),

(c) \( S' \) is consistent, where \( S' = (L, C, E \cup \{t = u\}) \).

**Proof.** (a) \( \iff \) (b): By Corollary 7.6

(a) \( \iff \) (c): By Lemma 9.1 and the definition of strong completeness. \( \square \)

10. Application to Abstract Data Types

In this section we present further examples of the definitions and theory of the previous sections. All of the examples will be systems that specify abstract data types. As already noted in Section 2, such specifications provided the original motivation for investigations into proof by consistency.

Let \( S = (L, C, E) \) be a system in which the language \( L \) is the union of languages \( L(\text{Bool}) \), \( L(\text{Nat}) \), and \( L(\text{Seq}) \), where \( L(\text{Bool}) \) is as described in Sections 5 and 6, \( L(\text{Nat}) \) is a language whose ground terms correspond to the set of natural numbers, and \( L(\text{Seq}) \) is a language whose ground terms correspond to finite sequences of natural numbers. We will not go into the details of \( L(\text{Nat}) \), but will discuss several versions of \( L(\text{Seq}) \) in this section. In each case we will have \( C \) consisting of \text{true}, \text{false}, and each of the natural number values. The function symbols belonging to \( \text{Seq} \) will include \text{null}, \( \dagger \), \( \div \), and \( \text{H} \), with arities:

\[
\text{null} : \rightarrow \text{Seq}, \\
\dagger : \text{Seq} \times \text{Nat} \rightarrow \text{Seq}, \\
\div : \text{Nat} \times \text{Seq} \rightarrow \text{Seq}, \\
\text{H} : \text{Seq} \times \text{Seq} \rightarrow \text{Seq}.
\]

The binary operators will be written in infix notation. We can read \( s \dagger x \) as "\( s \) with \( x \)," i.e., \( s \) with \( x \) appended to the right end of \( s \). Similarly, \( x \div s \) can be read "\( x \) onto \( s \)," i.e., the sequence obtained by prefixing \( x \) onto the left end of \( s \); while \( s \text{H} t \) denotes the concatenation of \( s \) and \( t \). (The notation and the flavor of the axiomatizations which follow are taken from Dahl [3].)
The first axiomatization we will consider is the following set of equations (i.e., $E$ consists of these equations in addition to those for $\text{Bool}$ and $\text{Nat}$):

\begin{align*}
  x \triangleright \text{null} &= \text{null} \triangleright x, \quad (1) \\
  (x \triangleright (s \triangleright y)) &= (x \triangleright s) \triangleright y, \quad (2) \\
  s \triangleright \text{null} &= s, \quad (3) \\
  s \triangleright (s \triangleright x) &= (s \triangleright s \triangleright x), \quad (4)
\end{align*}

Now, we will extend the above specification by adding new function symbols and axioms specifying them.

Case (i).

$\text{Empty?} : \text{Seq} \rightarrow \text{Bool}$;

with axioms:

\begin{align*}
  \text{Empty?}(\text{null}) &= \text{true}, \quad (5) \\
  \text{Empty?}(s \triangleright x) &= \text{false}. \quad (6)
\end{align*}

With equations (1)–(6), it can be shown that $S$ has a unique maximal $S$-congruence relation, in which there are two congruence classes of ground terms, one consisting of the single term $\text{null}$ and the other containing all other ground terms. Thus, $S$ is unambiguous.

The equation $\text{Empty?}(x \triangleright s) = \text{false}$ is consistent with these equations (as could be demonstrated by obtaining a canonical set of rewrite rules from $E \cup \{\text{Empty?}(x \triangleright s) = \text{false}\}$ under which $\text{true}$ and $\text{false}$ are not equated). Thus, by Theorem 9.2, $\text{Empty?}(x \triangleright s) = \text{false}$, where $=$ is the inductive congruence relation for $S$.

The following equation is inconsistent with (1)–(6):

\[(x \triangleright s) = s,\]

as from (1) and this equation, we deduce

\[(\text{null} \triangleright x) = \text{null},\]

which, with (6), gives us $\text{true} = \text{false}$. Thus, $(x \triangleright s) = s$ is not in the inductive theory of $S$. But the following equations are consistent with (1)–(6):

\begin{align*}
  ((s \triangleright x) \triangleright x) &= (s \triangleright x), \\
  ((s \triangleright x) \triangleright y) &= ((s \triangleright y) \triangleright x),
\end{align*}

and thus, by Theorem 9.2, are in the inductive theory of (1)–(6).
Consider another modification of $S$ in which the operation $\text{size}$ replaces $\text{Empty?}$.

*Case (ii).*

\[
\begin{align*}
\text{size} : \text{Seq} & \rightarrow \text{Nat} ; \\
\text{size}(\text{null}) & = 0 , \\
\text{size}(s\check{x}) & = \text{size}(s) + 1 .
\end{align*}
\]

(7)  
(8)

The system (1)-(4), (7)-(8) is again unambiguous. Having just the function $\text{size}$, we are able to distinguish among sequences of different sizes, but still not sequences of the same size. We have inconsistency with $(x\cdot s) = s$ as well as $((s\cdot x)\cdot x) = (s\cdot x)$, but $((s\cdot x)\cdot y) = ((s\cdot y)\cdot x)$ is still consistent. Thus, by unambiguity and Theorem 9.2, $(x\cdot s) = s$ and $((s\cdot x)\cdot x) = (s\cdot x)$ are not theorems of (1)-(4), (7)-(8), whereas $((s\cdot x)\cdot y) = ((s\cdot y)\cdot x)$ is in the inductive theory of (1)-(4), (7)-(8).

Let us consider yet another modification of $S$ in which to axioms (1)-(4) we add a new operation, $\text{rest}$, and its axioms, for obtaining the rest of a sequence after deleting its first element.

\[
\begin{align*}
\text{rest} : \text{Seq} & \rightarrow \text{Seq} ; \\
\text{rest}(\text{null}) & = \text{null} , \\
\text{rest}(\text{null}\check{x}) & = \text{null} , \\
\text{rest}(s\check{x}) & = \text{rest}(s\check{y})\check{x} .
\end{align*}
\]

(9)  
(10)  
(11)

Now if we also include $\text{Empty?}$ and (5)-(6), it is possible to distinguish among sequences of different sizes. The equation

\[
((s\cdot x)\cdot x) = (s\cdot x)
\]

would allow us to deduce

\[
\text{rest}(s\cdot x)\check{x} = \text{rest}(s\cdot x) ,
\]

which, with (11), would give

\[
(\text{rest}(s\cdot y)\cdot x) = \text{rest}(s\cdot y) ,
\]

and from (10) we would then have

\[
(\text{null}\cdot x) = \text{rest}(\text{null}\cdot y)\cdot x = \text{rest}(\text{null}\cdot y) = \text{null}
\]

from which we obtain inconsistency using (5) and (6), thus showing that the original equation is not a theorem.
However, we are still not able to distinguish among sequences of the same size. The following equation, for example, would still be consistent with (1)–(6), (9)–(11):

\[((s\uparrow x)\uparrow y) = ((s\uparrow y)\uparrow x)\].

We now add the operation \textit{first} to the axiomatization (1)–(6), (9)–(11).

\textit{first}: S \rightarrow N;

\begin{align*}
\text{first}(\text{null}\uparrow x) &= x, \\
\text{first}(s\uparrow y)\uparrow x) &= \text{first}(s\uparrow y). 
\end{align*}

(12) (13)

Axioms (1)–(6) and (9)–(13) can distinguish among unequal sequences. For example, adding \[((s\uparrow x)\uparrow y) = ((s\uparrow y)\uparrow x)\) would give an inconsistency.

11. Sufficient Completeness versus Unambiguity

In Section 10 we have been using the property of unambiguity of example systems, without going into detail as to how this property might be established in particular cases. Theorem 5.3 gives some characterizations of unambiguity that may be useful in establishing this property, e.g., showing that the possibly-equal relation, \sim, is transitive. In the case of abstract data types, the criterion of \textit{sufficient completeness} has frequently been used with purposes similar to the unambiguity property. In this section we consider the relation between sufficient completeness and unambiguity.

Let S be a system and \(L(T)\) be the language of some type \(T\) specified by S. The specification of \(T\) is said to be \textit{sufficiently complete} if for every ground term \(c\) in \(L(T)\) of type \(U\neq T\), there is some ground term \(c'\) of type \(U\) in \(L(U)\) such that \(c =_E c'\) [7].

\textbf{Theorem 11.1.} \textit{If a system S is consistent and every type specified by S is sufficiently complete, then S is unambiguous.}

\textbf{Outline of proof.} Kamin [15] has shown that under the hypotheses, S has a unique final model. As noted in Section 5 and Lemma 8.3, this final model corresponds to a unique maximal S-congruence relation. Hence S is unambiguous. \(\square\)

The converse of Theorem 11.1 is false. A similar observation was made by Kapur [16] and his example of a specification of sets is a counterexample to the implication that unambiguity implies sufficient completeness. The example is:
Example 11.2. To the axiomatization (1)–(6) add two more operations, Member? and Max, and their axiomatization (14)–(18):

\[
\begin{align*}
\text{Member?} & : \text{Seq} \times \text{Nat} \rightarrow \text{Bool}, \\
\text{Max} & : \text{Seq} \rightarrow \text{Nat}; \\
\text{Member?}(\text{null}, x) & = F, \\
\text{Member?}(s\uplus x, y) & = (\text{same}(x, y) \text{ or } \text{Member?}(s, y)), \\
\text{Member?}(s, \text{Max}(s)) & = \text{not}(\text{Empty?}(s)), \\
\text{Max}(\text{null}\uplus x) & = \text{max}(x, y), \\
\text{Max}((s\uplus w)\uplus x) & = \text{max}(\text{Max}((s\uplus w)\uplus x), y). \\
\end{align*}
\]

We are assuming that Nat has an equality function, same, and a function max which computes the maximum of two natural numbers.

Note that it is not the case that \(\text{Max}(\text{null}\uplus x) = E_x\) (even when \(x\) is a specific natural number value). In fact there is no natural number value that is in the \(=_E\) relation to \(\text{Max}(\text{null}\uplus x)\). Thus, the above specification is not sufficiently complete. On the other hand, the system is unambiguous. We will not show this in its entirety, but just give the proof that \(\text{Max}(\text{null}\uplus x)\) is uniquely determined. From (14) and (15) we have

\[
\text{Member?}(\text{null}\uplus x, \text{Max}(\text{null}\uplus x)) = \text{same}(x, \text{Max}(\text{null}\uplus x)),
\]

while from (16) we have

\[
\begin{align*}
\text{Member?}(\text{null}\uplus x, \text{Max}(\text{null}\uplus x)) \\
&= \text{not}(\text{Empty?}(\text{null}\uplus x)) = \text{not}(\text{false}) = \text{true}.
\end{align*}
\]

Thus,

\[
\text{same}(x, \text{Max}(\text{null}\uplus x)) = \text{true}
\]

and this implies that there is a unique value of \(y\), namely \(x\), for which the equation \(\text{Max}(\text{null}\uplus x) = y\) is consistent with the specification.

One way to resolve this difference between sufficient completeness and unambiguity might be to redefine sufficient completeness in terms of \(=\) instead of \(=_E\) (essentially this would allow inductive reasoning as well as equational reasoning in deducing the behaviour of constants):

A specification of a type \(T\) by a system \(S\) is called sufficiently complete (new definition) if and only if for every ground term \(c\) of type \(U \neq T\) in \(L(T)\), there is a ground term \(c'\) of type \(U\) in \(L(U)\) such that \(c = c'\).
Then a system would be unambiguous if and only if every type it specifies is sufficiently complete.

12. Sketch of an Extension to Ambiguous Systems

The method of proof by consistency depends upon the strong completeness property of the inductive congruence relation, which ensures that the border between theorems and nontheorems is the same as the border between consistency and inconsistency. In ambiguous systems these borders fail to coincide, leaving a gap filled by independent axioms. This is depicted in Fig. 7.

One might still wonder whether something like the method of proof by consistency could be devised to distinguish between theorems and independent axioms. The answer appears to be yes. In this section we briefly sketch an approach that we are taking to this problem; an interested reader may wish to look at [17] for further details. In the concluding section we discuss some of the implications that a solution to the problem would have.

The basic idea is to transform an ambiguous system into another unambiguous system, in which the theorems correspond to the theorems of the original system, and the inconsistent equations correspond to the union of the sets of independent axioms and inconsistent equations of the original system. That is, the border between consistency and inconsistency has been shifted so that the transformed system is strongly complete and one can use proof by consistency as a rule of inference to establish theorems, just as in the case of unambiguous systems. The transformation consists mainly of introducing a new completely defined function with extra arguments for every incompletely defined function;

Fig. 7. Pictorial representation of unambiguous and ambiguous systems.
these extra arguments are used to give all possible values to the function in those cases where the original function was not completely defined. The difficulties that arise in carrying out this transformation have to do with
(i) recognizing exactly on what arguments a function is not defined,
(ii) defining a predicate which holds for values that can serve as a possible result given by an incompletely defined function on a case on which it is not defined,
(iii) methods for enumerating the values a variable can take using the constructors of its type, and
(iv) dealing with the case which frequently arises that applying the Knuth–Bendix completion procedure to the transformed set of equations results in nontermination, as the only canonical set of rules would be infinite.

We are working on solutions to these problems and have encouraging preliminary results. Problem (iv) is an instance of a general problem, of canonical rewrite rule sets being infinite, to which it would be highly desirable to have a solution in many other useful cases (undecidability results preclude a completely general solution). The key to dealing with this problem appears to be the ability to recognize regular structure in infinite sets of rules and represent the rules by axiom schemas.

13. Conclusion

In this paper we have developed a general theory of proof by consistency and inductionless induction. The framework has been a general algebraic one, rather than the particular context of abstract data type specifications. Furthermore, when applied to abstract data types, the generality of our results means that it is possible to remove the various restrictions that previously had to be placed on the method, as discussed in Section 2, leaving only the requirement of unambiguity. As discussed in the preceding section, it appears that even this requirement can be removed.

If indeed the problems mentioned in the preceding section can be solved, it will be possible to build a data type specification system which gives considerably more assistance to the designer of a specification than is currently possible. In particular, as the designer inputs a new equation, the specification system could respond with one of four outcomes:
(i) the new equation results in inconsistency, with the system perhaps pointing out what is causing the inconsistency,
(ii) the new equation is a theorem, as it can be derived from the already existing axioms,
(iii) the new equation is independent of the existing axioms, and becomes part of the axiomatization,
(iv) the system responds with a trace of the generation of more and more
new equations, from which it is not possible to automatically determine which of cases (i)–(iii) hold.

In the last case, the user would have to direct the system to terminate this process, but might be able to guess from the structure of rules appearing in the trace what additional equations would lead to a decision.

In case (iii), it appears that the analysis that determines independence can also recognize the situation that, with the new axiom, the specification has been completed (has become strongly complete).

An equation intended by the user as a theorem might turn out to be an independent axiom, and vice versa. Note that it would be possible to find proofs of inductive properties in spite of ambiguity of the specification. This would permit specification methodologies in which specifications can be deliberately left ambiguous.

Appendix A

WELCOME TO REWRITE RULE LAB

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.

RRL → add
Type your equations, enter a "[" when done.
Equations read in are:
1. \( (x + 0) = = x \) [user, 1]
2. \( (x + s(y)) = = s((x + y)) \) [user, 2]
3. \( (0 * x) = = 0 \) [user, 3]
4. \( (s(x) * y) = = ((x * y) + y) \) [user, 4]
5. \( \text{sum}(0) = = 0 \) [user, 5]
6. \( \text{sum}(s(x)) = = (\text{sum}(x) + s(x)) \) [user, 6]
New constant set is: \( \{0\} \)
Time = 0.315 sec

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.

RRL → operator
Type abort, display, constructor, commutative, ac-operator, transitive, equivalence, precedence, status, manual, help → precedence
Type operators in decreasing precedence.
(e.g. "i*e" to set i > *> e)
\[ \text{sum} * + s \text{ 0} \]
Precedence relation, sum > *, is added.
Precedence relation, $*>+$, is added.
Precedence relation, $>s$, is added.
Precedence relation, $s>0$, is added.
Time = 0.1 sec

RRL $\rightarrow kb$

—Step 1—

Adding rule: [1] $(x + 0) \rightarrow x$ [user, 1]
Adding rule: [2] $(x + s(y)) \rightarrow s((x + y))$ [user, 2]
Adding rule: [3] $(0 * x) \rightarrow 0$ [user, 3]
Adding rule: [4] $(s(x) * y) \rightarrow ((x * y) + y)$ [user, 4]
Adding rule: [5] sum$(0) \rightarrow 0$ [user, 5]
Adding rule: [6] sum$(s(x)) \rightarrow s((sum(x) + x))$ [user, 6]

Computing critical pairs with:

[5] sum$(0) \rightarrow 0$ [user, 5]
Computing critical pairs with:

[6] sum$(s(x)) \rightarrow s((sum(x) + x))$ [user, 6]
Computing critical pairs with:

[3] $(0 * x) \rightarrow 0$ [user, 3]
Computing critical pairs with:

[1] $(x + 0) \rightarrow x$ [user, 1]
Computing critical pairs with:

[4] $(s(x) * y) \rightarrow ((x * y) + y)$ [user, 4]
Computing critical pairs with:

[2] $(x + s(y)) \rightarrow s((x + y))$ [user, 2]

Your system is canonical.

[1] $(x + 0) \rightarrow x$ [user, 1]
[2] $(x + s(y)) \rightarrow s((x + y))$ [user, 2]
[3] $(0 * x) \rightarrow 0$ [user, 3]
[4] $(s(x) * y) \rightarrow ((x * y) + y)$ [user, 4]
[5] sum$(0) \rightarrow 0$ [user, 5]
[6] sum$(s(x)) \rightarrow s((sum(x) + x))$ [user, 6]

Processor time used = 0.633 sec
Number of rules generated = 6
Number of critical pairs = 0
Time spent in normalization = 0.033 sec (5.263 percent of time)
Time spent while adding rules = 0.100 sec (15.789 percent of time)
(keeping rule set reduced)
Time spent in unification = 0.067 sec (10.526 percent of time)
Time spent in ordering = 0.050 sec (7.895 percent of time)
Total processor time used (include “undo” action) = 0.683 sec
Time = 1.017 sec

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.

RRL \rightarrow prove
Type equation to prove in the format: L = = R (if C)
Enter a “]” to exit when no equation is given.

\((x + y) = = (y + x)\) [user, 7]

No, it is not equational theorem.
Normal form of the left-hand side is:
\((x + y)\)
Normal form of the right-hand side is:
\((y + x)\)
Do you want to see it is an inductive theorem? y

—Step 2—
Current Constructor Set = {} 
To prove the equation with the constructors? (y, n) y 
Note: Constructor set is empty.
Type operators you wish to be constructors: 0 s
Constructor Set = {s, 0}

Specification of “sum” is complete relative to \{s, 0\}
Specification of “* ” is complete relative to \{s, 0\}
Specification of “+ ” is complete relative to \{s, 0\}
Processor time used = 0.117 sec
Time spent in reduction = 0.033 sec (28.571 percent of time)

Proving equation
\((x + y) = = (y + x)\) [user, 7]

—Step 3—
\((x + y) = = (y + x)\) [user, 7]
LHS of rule contains “+ ”, which is commutative now.
[1] \((x + 0) \rightarrow x\) [user, 1]
LHS of rule contains “+ ”, which is commutative now.
[2] \((x + s(y)) \rightarrow s((x + y))\) [user, 2]
RHS of rule contains “+ “, which is commutative now.
[4] \((s(x)*y) \rightarrow ((s*y) + y)\) [user, 4]
RHS of rule contains “+ “, which is commutative now.
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
PROOF BY CONSISTENCY

Adding rule: [7] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 2]
Adding rule: [8] \((x + 0) \rightarrow x\) [deleted, 1]
Computing critical pairs with:
[8] \((x + 0) \rightarrow x\) [deleted, 1]
Computing critical pairs with:
[7] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 2]

Your system is canonical.

[3] \((0 \times x) \rightarrow 0\) [user, 3]
[4] \((s(x) \times y) \rightarrow ((x \times y) + y)\) [user, 4]
[5] \(\text{sum}(0) \rightarrow 0\) [user, 5]
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
[7] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 2]
[8] \((x + 0) \rightarrow x\) [deleted, 1]

Following equation
\((x + y) = = (y + x)\) [user, 7]
is an inductive theorem in the current system.

Do you want to restore the previous system? (y, n) n

Processor time used = 0.67 sec
Time = 1.067 sec

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.

RRL→prove
Type equation to prove in the format: \(L = = R\) (if C)
Enter a '"!' to exit when no equation is given.
\((x + (y + z)) = = ((x + y) + z)\) [user, 8]

No, it is not equational theorem.
Normal form of the left-hand side is:
\((x + (y + z))\)
Normal form of the right-hand side is:
\(((x + y) + z)\)

Do you want to see it is an inductive theorem? y
—Step 4—
Proving equation
\((x + (y + z)) = = ((x + y) + z)\) [user, 8]
—Step 5—
\((x + (y + z)) = = ((x + y) + z)\) [user, 8]
RHS of rule contains " + ", which is AC now.
[4] \((s(x) \times y) \rightarrow ((x \times y) + y)\) [user, 4]
RHS of rule contains " + ", which is AC now.
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
LHS or rule contains "+", which is AC now.
[7] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 2]
LHS of rule contains "+", which is AC now.
[8] \((x + 0) \rightarrow x\) [deleted, 1]

Adding rule: [9] \((x + 0) \rightarrow x\) [deleted, 8]
Adding rule: [10] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 7]

Your system is hopefully canonical (Current RRL supports no AC-ordering).

[3] \((0 \ast x) \rightarrow 0\) [user, 3]
[4] \((s(x) \ast y) \rightarrow ((\ast y) + y)\) [user, 4]
[5] \(\text{sum}(0) \rightarrow 0\) [user, 5]
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
[9] \((x + 0) \rightarrow x\) [deleted, 8]
[10] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 7]

Following equation
\((x + (y + z)) = = ((x + y) + z)\) [user, 8]
is an inductive theorem in the current system.
Do you want to restore the previous system? (y, n)

Processor time used = 0.90 sec
Time = 1.05 sec

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.
RRL → prove
Type equation to prove in the format: L = = R (if C)
Enter a "[" to exit when no equation is given.

\((x \ast y) = = (y \ast x)\) [user, 9]

No, it is not equational theorem.
Normal form of the left-hand side is:
\((x \ast y)\)
Normal form of the right-hand side is:
\((y \ast x)\)
Do you want to see it is an inductive theorem? y

— Step 6 —
Proving equation
\((x \ast y) = = (y \ast x)\) [user, 9]

— Step 7 —
\((x \ast y) = = (y \ast x)\) [user, 9]
LHS of rule contains "\ast", which is commutative now.
[3] \((0 \ast x) \rightarrow 0\) [user, 3]
Proof by Consistency

LHS of rule contains "\*", which is commutative now.

[4] \((s(x) \ast y) \rightarrow ((x \ast y) + y)\) [user, 4]

Adding rule: [11] \((s(x) \ast y) \rightarrow ((x \ast y) + y)\) [deleted, 4]
Adding rule: [12] \((0 \ast x) \rightarrow 0\) [deleted, 3]

Your system is hopefully canonical (Current RRL supports no AC-ordering).

[5] \(\text{sum}(0) \rightarrow 0\) [user, 5]
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
[9] \((x + 0) \rightarrow x\) [deleted, 8]
[10] \((x + s(y)) \rightarrow s((x + y))\) [deleted, 7]
[11] \((s(x) \ast y) \rightarrow ((x \ast y) + y)\) [deleted, 4]
[12] \((0 \ast x) \rightarrow 0\) [deleted, 3]

Following equation
\(x \ast y = (y \ast x)\) [user, 9]
is an inductive theorem in the current system.
Do you want to restore the previous system? (y, n) n

Processor time used = 0.72 sec
Time = 0.917 sec

Type Add, Akb, Auto, Break, Clean, Delete, Dump, Grammar, Init, Kb, List, Log, Norm, Order, Option, Operator, Prove, Quit, Read, Refute, Stats, Suffic, Undo, Unlog, Write or Help.

RRL -> prove
Type equation to prove in the format: \(L = = R\) (if C)
Enter a "\]" to exit when no equation is given.
\(s(s(0)) \ast \text{sum}(x) = = ((x \ast x) + x)\) [user, 10]

No, it is not equational theorem.
Normal form of the left-hand side is:
\(\text{sum}(x) + \text{sum}(x)\)
Normal form of the right-hand side is:
\((x \ast x) + x\)
Do you want to see it is an inductive theorem? y
—Step 8—

Proving equation
\(s(s(0)) \ast \text{sum}(x) = = ((x \ast x) + x)\) [user, 10]

Adding rule: [13] \((\text{sum}(x) + \text{sum}(x)) \rightarrow (x + (x \ast x))\) [user, 10]

Your system is hopefully canonical (Current RRL supports no AC-ordering).

[5] \(\text{sum}(0) \rightarrow 0\) [user, 5]
[6] \(\text{sum}(s(x)) \rightarrow s((\text{sum}(x) + x))\) [user, 6]
[9] \((x + 0) \rightarrow x\) [deleted, 8]
Following equation

\[(s(s(0)) * \text{sum}(x)) = (x * x + x)\]  

is an inductive theorem in the current system.

Do you want to restore the previous system? (y, n)

Processor time used = 2.53 sec
Time = 2.817 sec

Note. The termination of rewriting systems involving associative/commutative operators cannot be proved by RRL; however the termination of such rewriting systems in the above example can be easily established by hand.

REFERENCES


Received June 1984; revised version received April 1986