ON PROVING INDUCTIVE PROPERTIES OF ABSTRACT DATA TYPES

David R. Musser
USC Information Sciences Institute

Abstract

The equational axioms of an algebraic specification of a data type (such as finite sequences) often can be formed into a convergent set of rewrite rules; i.e. such that all sequences of rewrites are finite and uniquely terminating. If one adds a rewrite rule corresponding to a data type property whose proof requires induction (such as associativity of sequence concatenation), convergence may be destroyed, but often can be restored by using the Knuth-Bendix algorithm to generate additional rules. A convergent set of rules thus obtained can be used as a decision procedure for the equational theory for the axioms plus the property added. This fact, combined with a "full specification" property of axiomatizations, leads to a new method of proof of inductive properties--not requiring the explicit invocation of an inductive rule of inference.

This work was supported by the Defense Advanced Research Projects Agency under contract No. DAHC15 72 C 0308. The views expressed are those of the author.

Author's present address: Computer Science Branch, General Electric Research and Development Center, Schenectady, New York 12345.

1. INTRODUCTION

In the algebraic axioms method of specification of abstract data types [3,7,8,9,10,11,12,13,14,29], the semantic properties of a collection of functions are described by a set of equations relating the functions to each other. A principal advantage of this specification technique is that often one can obtain a decision procedure for the theory of the equations and use it to automatically carry out proofs of properties of the data type (or, at least, many steps of proofs). An important way of obtaining such decision procedures is to form the equations into rewrite rules and use the methods Knuth and Bendix [18] devised for equational theories of mathematical structures such as noncommutative group theory. The application of rewrite rule techniques to the proof theory of data types is more complex than to structures such as groups and rings, because of an additional, non-equational rule of inference assumed for data types--an induction principle, called *generator induction* [26] or *data type induction* [12]. Roughly speaking, this induction principle states that a formula about the functions of the type is a theorem if it can be shown to hold for all "constants" of the type, where a constant is an expression finitely constructable by composition of the function symbols belonging to the type (thus it is a ground expression, i.e., it contains no variables). The usual way of applying this induction principle has been to perform "structural induction" on the structure of constants, or induction based on some other well-founded partial ordering of the constants. Such proofs require explicit invocation of a suitable induction principle for each formula to be proved, and thus there has been no way to carry out these proofs working just with equations. The main new result described in this paper is a theorem showing that by observing certain constraints on the axiomatizations of data types, the use of induction becomes "meta" and proofs of inductive data type properties often can be carried out without

explicitly invoking an induction principle. This method of proof, in which the Knuth-Bendix methods play an integral role, has been implemented in an experimental program verification system at the USC Information Sciences Institute, the AFFIRM system [23]. Experience with the method has shown that in many cases it can simplify considerably the task of proving inductive properties of data types.

Before presenting the main theorem we attempt in the following section to provide clear and concise definitions of many of the basic concepts relating to the proof theory of algebraically specified data types, including "the language of a type," "equational theory," "consistency," and "inductive theory." The main requirement placed on specifications in order to permit the application of the main theorem is a kind of completeness requirement called "full specification." It is closely related to "sufficient completeness" [14], but has a simpler definition. A lemma is proved in Section 3 that shows that rewrite rule techniques can be used to prove this property about collections of data type specifications.

Courcelle [4] has (independently) proved a result that appears to be similar to our main theorem, but in a recursive function theory setting. Among previous efforts to apply rewrite rule techniques to proving data type properties, the results reported by Bledsoe and Bruell [1], Boyer and Moore [2], von Henke and Luckham [15], Lankford [19], and Suzuki [28] have been important sources of inspiration. For an introduction to basic rewrite rule concepts, particularly the "convergence property" (finite and unique termination of all sequences of rewrites), see [23]. For a broader discussion of the AFFIRM system and related work by other authors, again see [23].

2. ABSTRACT DATA TYPES: BASIC CONCEPTS

Specifications

An *abstract data type specification* consists of a name T; a syntactic part, specifying the language of the type; and a semantic part, specifying the axioms which the expressions of the language satisfy. We will use the simple phrase "type T" in place of "abstract data type specification for T."

The syntactic part of the specification lists a set of function symbols $f_1,...,f_n$ belonging to the type and for each f_i the names of its argument types and range type (the *arity* of f_i). One or more *initial functions* must be included in the syntactic specification. These are functions which have range type T but have no arguments of type T. For each of the type names T' that appear in the arities, other than T itself, we say that T *depends directly* on T'. For T and U in the reflexive, transitive closure of this relation, we say that T *depends on* U.

We define the *language of T*, written *L(T)*, and a mapping *TypeOf* from expressions to type names, inductively as follows: (a) For each type U on which T depends, an infinite set of variable symbols v, with TypeOf(v) = U, are contained in L(T). (b) For each type U on which T depends, and each function f belonging to U, let the arity of f be $U_1 \times ... \times U_k \rightarrow U_{k+1}$ where $k \geq 0$; then for each tuple $(e_1,...,e_k)$ of expressions e_i in L(T) with $TypeOf(e_i) = U_i$ for $i=1,...,k$, the expression $f(e_1,...,e_k)$ is in L(T) with $TypeOf(f(e_1,...,e_k)) = U_{k+1}$. (c) No other expressions are in L(T).

This definition of L(T) differs from that in Guttag and Horning [14] in including variable symbols, but agrees with a more recent definition in [11]. Note that it permits mutual dependency among specifications, since the inductive definition is based on the structure of expressions, rather than on a hierarchy in the organization of specifications.

Turning to the semantic part of the specification, the *axioms* of T are a finite set of equations in L(T). It is assumed that among the operators belonging to any type T is the operator "$=_T$" used to express the axioms. The arity of "$=_T$" is $T \times T \rightarrow$ Boolean. (Type Boolean is discussed below.) These equality symbols will usually be written without the subscript T, since T can be inferred as the type of the arguments. The axioms are assumed to include the reflexive property for $=_T$, i.e. $(x =_T x) =_{Boolean}$ true. (Discussion of other requirements on equality operators appears in footnote 2.)

If T is a collection $\{T_1,...,T_n\}$ of types, the language of T, again denoted L(T), is the union of $L(T_i)$, $i=1,...,n$, and the axiom set A of T is the union of the axiom sets A_i of T_i, $i=1,...,n$.

These definitions are illustrated in the following subsection and in the specification of type SequenceOfElement given later (Figure 1).

The Boolean type

The following is a specification of a type named Boolean, used to express logical values. It is expressed in the syntax accepted by the AFFIRM system.

```
type Boolean;
declare p,q,r: Boolean;
interface  true,
           false,
           if p then q else r,
           not(p),
           p and q,
           p or q,
           p implies q  : Boolean;
axiom
  [p = q) = if p then q else not(q),
  (p = p) = true,

  (if true then p else q)  =  p,
  (if false then p else q)  =  q,

  not(p)  =  if p then false else true,

  (p and q)  =  if p then q else false,

  (p or q)  =  if p then true else q,

  (p implies q)  =  if p then q else true;

end (Boolean) ;
```

Notes: p,q,r are declared to be variables with TypeOf(p)=TypeOf(q)=TypeOf(r)=Boolean. The symbols "true" and "false" are constant initial functions (actually true() and false(), but AFFIRM permits dropping the "()" on input); "if p then q else r" is a more convenient syntax for if-then-else(p,q,r), "p and q" is the infix form of and(p,q), etc. The arity of "and" is Boolean x Boolean -> Boolean, etc. This kind of axiomatization of propositional calculus, in terms of an if-then-else operator, is similar to one suggested by McCarthy and used by Boyer and Moore.

Type Boolean does not depend on any other type, but every type depends on Boolean (since, at least, "$=_T$" has range Boolean).

Predicate expressions are expressions P for which TypeOf(P) is Boolean. Thus all expressions in L(Boolean) are predicate expressions, as are all equations in any L(T). Variables in predicate expressions that are axioms or theorems are implicitly universally quantified; no explicit quantifiers are permitted.

Equational theory

Let A be a finite set of equations $\{\alpha_i=\beta_i\}$. We have, for each equation $\alpha_i=\beta_i$, a set of equations obtained by allowing the variables of α_i and β_i to range over the set of all expressions (this set is infinite unless both α and β are constant expressions). This defines a binary relation "=" on expressions, and we extend "=" to the smallest equality relation[1] containing all of the equations generated. We write $\alpha=\beta$ (by A) for the resulting equality relation, and call the set of equations satisfying this relation the *equational theory of A*. If T is a type or a collection of types with axiom set A, then the equational theory of T is defined to be the intersection of the equational theory of A with L(T).[2]

Constants

The *constants* of a type or collection of types, T, are the ground expressions of L(T), i.e., those expressions in L(T) containing no variable symbols. A set of *canonical constants* for a type T with axiom set A is a set of representatives of the equivalence classes of constants under the equality relation $\alpha = \beta$ (by A).

With the specification of type Boolean given above, it can be shown that each constant of L(Boolean) is equal (by the axioms of Boolean) to one of the symbols "true" or "false," but "true=false" does not follow from the axioms, and thus the set of canonical constants of type Boolean can be taken to be {true,false}.

Consistency

A set of equations is *consistent* if "true=false" is not in its equational theory. A type or collection of types T with axioms A is defined to be consistent if A is consistent. For example, the Boolean type given above is consistent.

Fully specified types

Let T be a collection of types with axiom set A. T is *fully specified (by A)* if every constant c in L(T) is equal (by A) to a constant in L(TypeOf(c)).

The full specification property is closely related to the "sufficient completeness" property of single type specifications [14]. Its significance lies primarily in that if one has a collection of types that has been shown to be fully specified, and one extends the collection with a new type specification so that the augmented collection

---

[1] An equality relation (or, more frequently, congruence relation) is an equivalence relation that also has the property of being preserved under substitution.

[2] Note that the axioms given for the equality operator of a type must be checked for consistency with the equivalence and substitution properties. The substitution property can be checked by confirming that for each operator of the type, substitution of equal arguments yields equal values; e.g., q=q1 and i=i1 implies Add(q,i) = Add(q1,i1). If the axioms for equality are also consistent with the reflexive property ($x =_T x$), then it can be shown that the symmetric property ($x =_T y$ implies $y =_T x$), and the transitive property ($x =_T y$ and $y =_T z$ implies $x =_T z$), also hold.

is also fully specified, then the added specification does not introduce any new constants into the old types.

Lemma (Reduction of constant predicates). Let T be a collection of types with axiom set A, and assume that T is fully specified by A. If P_0 is a predicate constant in L(T), then either P_0 = true (by A) or P_0 = false (by A).

Proof. From the three facts (1) P_0 is a constant, (2) TypeOf(P_0) is Boolean, and (3) T is fully specified by A, we can conclude that there is a constant P_1 in L(Boolean) such that $P_0 = P_1$ (by A). Since the set of canonical constants of type Boolean is {true, false}, either P_1 = true (by A) or P_1 = false (by A); hence P_0 = true (by A) or P_0 = false (by A). QED.

Inductive theory

Let T be a type or a collection of types with axiom set A. The *inductive theory* of T is the set of all predicate expressions P such that every constant P_0, obtained by substituting constants for the variables of P, satisfies P_0=true (by A).

Thus the inductive theory of T is the theory obtained by adding to the equational theory A of T an inductive rule of inference, of the following form (let $P=Q(x_1,...,x_n)$ be a predicate expression in L(T)):

$$\frac{\begin{array}{c}A \vdash Q(c_1,...,c_n)=\text{true} \\ \text{for all tuples } (c_1,...,c_n) \text{ of constants} \\ \text{such that TypeOf}(c_i) = \text{TypeOf}(x_i)\end{array}}{\forall x_1,...,x_n,\ Q(x_1,...,x_n)}$$

A rule of inference of this form is sometimes called a "rule of infinite induction," since there are infinitely many premises to the rule. As such, it is not directly applicable in proofs as a rule of inference, but it can be replaced by other rules that have only finitely many premises corresponding to the structure of the constants--structural induction, or, more generally, induction based on any well-founded partial ordering of the constants. Such induction rules have been called *generator induction* [26] or *data type induction* [12]. These usual methods of inductive proof of data type properties are closely related to the structural induction methods used by Boyer and Moore to prove properties of recursive functions, but the method of proof to be described in the next section is different from the Boyer-Moore techniques.

In the initial algebra approach of [3,8], the inductive theory of T corresponds to the initial algebra of T.

3. FULL SPECIFICATION PLUS EQUATIONAL DECIDABILITY YIELD INDUCTIVE PROOF

Theorem. Let T be a collection of types with axiom set A, and assume that T is fully specified by A. If E is a set of equations, each in L(T), and A ∪ E is consistent, then each equation in E is in the inductive theory of T.

Proof. Let $\alpha=\beta$ be an equation in E, and $\alpha_0=\beta_0$ be any constant instance of $\alpha=\beta$. We must show that A $\vdash$ $(\alpha_0=\beta_0)$=true. By the lemma, either A $\vdash$ $(\alpha_0=\beta_0)$=true or A $\vdash$ $(\alpha_0=\beta_0)$=false. Suppose the latter were the case, then A ∪ E $\vdash$ $(\beta_0=\beta_0)$=false (using $\alpha=\beta$ to replace α_0 by β_0), and A ∪ E $\vdash$ true=false (by the assumption that A contains the reflexive axiom for $=_U$ where U=TypeOf(β_0)), contradicting the assumed consistency of A ∪ E. Thus, A $\vdash$ $(\alpha_0=\beta_0)$=true. QED.

This theorem shows that the Knuth-Bendix algorithm can be used to prove inductive properties: suppose one has a collection of data type specifications, their axiomatizations have previously been formed into a convergent set of rewrite rules,[3] and it has been checked that the types are fully specified by their axioms. To attempt to prove that an equation $\alpha=\beta$ is in the inductive theory of the collection of types, one adds a new rewrite rule (either $\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$ according to the finite termination criterion being used) and performs the Knuth-Bendix algorithm. There are three possible outcomes. First, the algorithm may terminate after generating a finite number (possibly zero) of additional rules, none of which is "true → false," with the convergence property affirmed. In this case, according to our theorem, the equation $\alpha=\beta$ is a theorem. Note how this depends on the capability of deciding equations: we are certain that true=false is not a consequence of the augmented equations because both true and false are irreducible constants. Second, the rule "true → false" may be generated, in which case the equation $\alpha=\beta$ is inconsistent with the axioms of the types. The third possibility is that the Knuth-Bendix algorithm may not be able to find a finite convergent set of rules, and continues generating rules indefinitely. (Thus, some other provision must be made for terminating the algorithm, such as a limit on number of newly generated rules or time consumed. In the AFFIRM system, the user sees the generated rules being printed on the terminal and can interrupt the process manually.) In this case, no definite information is gained about whether $\alpha=\beta$ is a theorem, although the structure of the rules generated before the

[3] A set of rewrite rules is defined to be convergent [23] if it has both the finite and unique termination properties. A fundamental property of convergent sets of rules is that they provide a decision procedure for the equational theory of the equations to which they correspond.

process is stopped may be suggestive of additional equations that would lead to convergence. This situation is similar to behavior of the Boyer-Moore theorem prover when it sometimes needs a lemma to be supplied by the user before it can find a proof of a theorem. Each of these outcomes is illustrated in the examples that follow.

Examples: inductive properties of sequences

Figure 1 shows a specification of finite sequences, based on material presented by Dahl in [5]. It can be shown that when the equations of this specification are treated as rewrite rules they are convergent; the proof of the unique termination property is carried out by the AFFIRM system using the Knuth-Bendix algorithm, when the equations are fed into the system. Furthermore, the collection of types containing SequenceOfElement, Element, and Boolean can be shown to be fully specified, by application of a lemma to be proved at the end of this section.

Before examining how our theorem permits the Knuth-Bendix algorithm to be used to prove inductive properties about sequences, let us first consider how such properties would be proved by a more conventional approach. According to the interface specifications and the principle of data type induction, the values of this type can be regarded as the set of all constant expressions finitely constructable using the null, +, and onto functions. Thus proofs of inductive properties could be attempted using the following rule of inference:

$$\frac{\begin{array}{c}P(null)\\ P(s) \text{ implies } P(s + x)\\ P(s) \text{ implies } P(x \text{ onto } s)\end{array}}{P(s)}$$

If one first uses this rule of inference to prove a "normal form lemma" for the type, showing that occurrences of the "onto" operator can always be eliminated from constants, then the third premise of the above rule can be omitted in all subsequent proofs of inductive properties. Thus, to prove the property (s cat (s' + x') = (s cat s') + x'), for example, one can take P(s) = ∀s',x'[s cat (s' + x') = (s cat s') + x'], and it suffices to prove (1) P(null), and (2) P(s) implies P(s + x).

The proof of P(null) simply requires two applications of the axiom [null cat s = s]. Two applications of the axiom [(s + x) cat s' = s cat (x onto s')] reduce (2) to [P(s) implies s cat (x onto (s' + x')) = (s cat (x onto s')) + x'], to which the axiom [x onto (s + x') = (x onto s) + x'] applies, yielding [P(s) implies s cat ((x onto s') + x') = (s cat s') + x']. We may then use the induction hypothesis, P(s), by renaming bound variables s' to s'' and x' to x'', and taking s'' = (x onto s') and x'' = x' to reduce the conclusion to an identity.

Figure 1
Element and SequenceOfElement Types

```
type Element;

declare x, x': Element;

interface  errElement: Element;

axiom   (x=x)  =  true;

end (Element) ;


type SequenceOfElement;

declare  s, s', s'',s''': SequenceOfElement;
declare x, x', x'': Element;

interface
  null,    {the sequence of zero elements}
  s+x,     {the sequence obtained by adding x
                to the end of s}
  x onto s, {the sequence obtained by adding x
                  to the front of s}
  s cat s'  {the sequence obtained by
                  concatenating s and s'}
  : SequenceOfElement;

interface
  isnull(s)
  : Boolean;

axiom
  (s=s)  =  true,
  (null = s+x)  =  false,
  (s+x = null)  =  false,
  (s+x = s'+x')  =  ((x=x') and (s=s')),

  x onto null  =  null + x,
  x onto (s+x')  =  (x onto s) + x',

  null cat s  =  s,
  (s+x') cat s'  =  s cat (x' onto s'),

  isnull(null)  =  true,
  isnull(s+x)  =  false;

end (SequenceOfElement) ;
```

Note that this proof required several nontrivial steps: (1) choice of induction variable (if we had chosen s' instead of s, then even the basis case would not have been directly provable); (2) application of the appropriate axioms to make equational substitutions, and (3) renaming bound variables and choosing an appropriate instantiation in applying the induction hypothesis

Now consider how the same inductive property would be proved using the Knuth-Bendix algorithm. To the original set of rules obtained from the axioms, one adds the rule [s cat (s' + x') → (s cat s') + x']. It must be checked that the augmented set of rules still has the finite termination property. When the Knuth-Bendix algorithm is then performed, it shows that the unique termination property also still holds; no additional rules are required to achieve convergence. Since the collection of types has the full specification property, our theorem implies that the property is in the inductive theory of this collection of types. (Readers familiar with the Knuth-Bendix algorithm may wish to examine how the construction of superpositions and critical pairs corresponds fairly directly to the main steps of the standard inductive proof. The main difference is that one has, in effect, a stronger induction hypothesis, since the induction can be regarded as being on the maximum length of sequences of rewrites of an expression.)

Now suppose one wishes to prove the inductive property [s cat null = s]. Again, when this is added to the rules consisting of the axiom rules and the rule [s cat (s' + x') → (s cat s') + x'], the convergence property is affirmed by the Knuth-Bendix algorithm. If, however, one considers the equation [s cat null = null], and adds this as a rule, the Knuth-Bendix algorithm quickly generates the rule [true → false], showing that [s cat null = null] is inconsistent with the axioms. This illustrates the second of the three possible outcomes mentioned above.

The third possibility is simply illustrated by adding the rule [s cat null = s] to the axioms alone, without the rule [s cat (s' + x') → (s cat s') + x']. The Knuth-Bendix algorithm begins generating the sequence of rules

s cat (null + x) → s + x
s cat ((null + x) + x') → (s + x) + x'
s cat (((null + x) + x') + x'') → ((s + x) + x') + x''
etc.

It is easy to see from the form of these rules that the rule [s cat (s' + x') → (s cat s') + x'] would prevent them from being formed since it permits the left side of each to be reduced to the right side. Thus one must have first added this "associativity property," or it must be added simultaneously with [s cat null → s].

Another interesting example is associativity of "cat": [s cat (s' cat s'') = (s cat s') cat s'']. When this is added as a rule, the Knuth-Bendix algorithm generates two rules that appear to be the beginning of an infinite sequence of rules,

(s cat (x' onto s')) cat s''
    → s cat (x' onto (s' cat s''))
s cat (x onto ((x' onto s') cat s''))
    → s cat (x onto (x' onto (s' cat s')))

However, it then generates another associativity rule, [(x onto s) cat s' → x onto (s cat s')], which allows the second of the two rules above to be discarded, and prevents generation of further similar rules; the resulting set of rules is found to be convergent. Thus, in this case, the "lemma needed to complete the proof" was generated automatically.

Additional functions can be added to the sequence type, and many other inductive properties can be proved by this method. The main limitation is the difficulty of proving the finite termination property of the rules generated. Figure 2 shows additional functions and some of the properties that have been shown to hold, assuming finite termination, using the Knuth-Bendix algorithm in the AFFIRM system.

Extensions

The Theorem ought to be generalized to permit parameterized type specifications. We are studying the proposals of Burstall and Goguen [3] and Nakajima [24] for specification languages that include type parameterization. Another area for extension is to study the effects of including exception handling in the specifications, in the manner presented by Guttag in [11].

Another issue is that of the practicality of meeting the requirement of "full specifications." Considerable experience with algebraic specifications gained by the author and others indicates that this is not a severe requirement. The restrictions on the form of axioms that Guttag and Horning [14] have given for guaranteeing the related condition of "sufficient completeness," however, appear to be stronger than necessary. A less restrictive approach is possible, based on the following

Lemma. Let T be a collection of types with axioms A. Suppose the axioms of A form a set of rewrite rules with the finite termination property, and under this set of rules every constant c in L(T) is either reducible or is in L(TypeOf(c)). Then T is fully specified by A.

Proof. We must show that any constant c in L(T) is equal (by A) to a constant in L(TypeOf(c)). If c is not already in L(TypeOf(c)), then by assumption it is reducible, to some

Figure 2
Extended SequenceOfElement Type

```
type SequenceOfElement;

declare s, s', s'',s''': SequenceOfElement;
declare x, x', x'': Element;

interface
  null,
  s+x,
  x onto s,
  s cat s',
  butfirst(s),
  butlast(s),
  errSeq,
  reverse(s)
  : SequenceOfElement;

interface
  isnull(s),
  x in s,
  noduplicates(s),
  s disjointfrom s'
  : Boolean;

interface
  first(s),
  last(s)
  : Element;

axiom
  (s=s)  =  true,
  (null = s+x)  =  false,
  (s+x = null)  =  false,
  (s+x = s'+x')  =  ((x=x') and (s=s')),

  errSeq  =  null,

  x onto null  =  null + x,
  x onto (s+x')  =  x onto s + x',

  null cat s  =  s,
  (s+x') cat s'  =  s cat (x' onto s'),

  isnull(null)  =  true,
  isnull(s+x)  =  false,

  first(null)  =  errElement,
  first(s+x)  =  if isnull(s) then x else first(s),

  last(null)  =  errElement,
  last(s+x)  =  x,

  butfirst(null)  =  errSeq,
  butfirst(s+x)  =
    if isnull(s) then null else butfirst(s) + x,

  butlast(null)  =  errSeq,
  butlast(s+x)  =  s,

  x in null  =  false,
  x in (s+x')  =  (not (x=x') implies x in s),

  reverse(null)  =  null,
  reverse(s+x)  =  x onto reverse(s),

  noduplicates(null)  =  true,
  noduplicates(s+x)  =
    (not (x in s) and noduplicates(s)),

  null disjointfrom s'
      =  true,
  (s+x) disjointfrom s'
      =  (not (x in s') and s disjointfrom s');

rulelemma
  s cat (s'+x')  =  (s cat s') + x',
  s cat null  =  s,
  s cat (s' cat s'')  =  (s cat s') cat s'',
  (s cat (x' onto s')) cat s''
      =  s cat (x' onto (s' cat s'')),
  (x' onto s') cat s''
      =  x' onto (s' cat s''),

  isnull(x onto s)  =  false),
  isnull(s cat s')  =  (isnull(s) and isnull(s')),

  first(x onto s)  =  x,

  last(x onto s)
    =  if isnull(s) then x else last(s),

  butfirst(x onto s)  =  s,

  butlast(x onto s)  =
    if isnull(s) then null else x onto butlast(s),

  x in (x' onto s)  =  (not (x=x') implies x in s),
  x in (s cat s')
    =  (not (x in s) implies x in s'),

  reverse(x onto s)  =  reverse(s) + x,
  reverse(s cat s')  =  reverse(s') cat reverse(s),

  noduplicates(x onto s)
      =  (not (x in s) and noduplicates(s)),

  s disjointfrom null
      =  true,
  s disjointfrom (s'+x')
      =  (not (x' in s) and s disjointfrom s');

end {SequenceOfElement} ;
```

constant c_1 in L(T). We continue in this way, obtaining a sequence $c, c_1, c_2, \ldots$, eventually reaching a constant c_n that is irreducible (since the rules have the finite termination property), and thus is in L(TypeOf(c_n)). Then $c=c_n$ (by A) and TypeOf(c_n)=TypeOf(c). QED.

Based on this lemma, the full specification property can be proved by first proving the finite termination property, then analyzing the reducibility of constants.

An important extension that needs to be made in the AFFIRM implementation is to provide algorithmic checks for sufficient conditions for termination (finite termination of arbitrary sets of rewrite rules is undecidable [17,21]), rather than leaving it up the user to deal with termination outside the system, as is currently done. Among the possible methods for proving termination being considered are those described in [6] and [22].

It would also be useful to implement some of the generalizations of the Knuth-Bendix methods devised by Huet [16], Lankford and Ballantyne [20], Nelson and Oppen [25], and Stickel and Peterson [27], to handle larger classes of equations, including many cases of commutative and associative axioms.

4. CONCLUSION

The method of proof described in this paper has been used experimentally in the AFFIRM system to prove numerous inductive properties of data types such as sequences, queues, sets, circular lists, and trees. The theorems about sequences shown in Figure 2 are fairly typical of the kinds of properties proved about these types.

There are some kinds of useful data type properties for which our proof method is not applicable, e.g. transitivity of the subset relation about sets. The difficulty that arises is that expression of the property as a rewrite rule requires use of the if-then-else operator, and although there are cases in which convergent sets of rules involving this operator can be obtained, most often the Knuth-Bendix algorithm keeps on generating longer and longer rules and does not terminate.

For most inductive properties that can be expressed as simple equations, however, our method of proof has usually been applicable. Except for the proof of finite termination, for which AFFIRM as yet provides no assistance, this has permitted the proofs of such properties to be obtained much more easily than when such proofs are done by explicitly invoking an inductive rule of inference.

Acknowledgments

It is a pleasure to acknowledge helpful discussions with Bob Boyer, Rod Erickson, Susan Gerhart, Joe Goguen, John Guttag, Jim Horning, Dallas Lankford, Ralph London, J Moore, Derek Oppen, and David Thompson. Their suggestions have helped in many ways to clarify the ideas presented. Special thanks are due to John Guttag for valuable comments on earlier drafts of this paper.

REFERENCES

1. Bledsoe, W. W., and P. Bruell, "A Man-Machine Theorem-Proving System," *Artificial Intelligence*, Vol. 5, pp. 51-72 (1974).

2. Boyer, R. S., and J S. Moore, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory," *Proceedings IJCAI-77 Conference*, Vol. 1, pp. 511-519 (August 1977).

3. Burstall, R. M. and J. A. Goguen, "Putting Theories Together to Make Specifications," *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, August 1977, pp. 1045-1058.

4. Courcelle, B., "On Recursive Equations Having a Unique Solution," IRIA-LABORIA Report No. 285, March 1978.

5. Dahl, O. J., "Can Program Proving Be Made Practical?" Institute of Informatics, University of Oslo, Norway, (1978).

6. Dershowitz, N., and Z. Manna, "Proving Termination with Multiset Orderings," Computer Science Department Report No. STAN-CS-78-651, Stanford University, March 1978.

7. Goguen, J.A., and J.J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Specifications," *Proceedings of Specification of Reliable Software Conference* Boston, April 3-5, 1979, pp. 170-189.

8. Goguen, J.A., J.W. Thatcher, E.G. Wagner and J.B.Wright, "Abstract Data Types as Initial Algebras and the Correctness of Data Representations," *Proceedings of Conference on Computer Graphics, Pattern Recognition and Data Structure*, Beverly Hills, Ca., pp. 89-93 (1975).

9. Guttag, J. V., "The Specification and Application to Programming of Abstract Data Types," Ph. D. Thesis, University of Toronto, Department of Computer Science, 1975.

10. Guttag, J. V., "Abstract Data Types and the

Development of Data Structures," *Communications of the ACM*, Vol. 20, June 1977, pp. 397-404.

11. Guttag, J. V., "Notes on Type Abstraction," *Proceedings of Specifications of Reliable Software Conference*, Boston, April 3-5, 1979, pp. 36-46. Also to appear in *IEEE Transactions on Software Engineering*.

12. Guttag, J. V., E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM*, Vol. 21, December 1978.

13. Guttag, J. V., E. Horowitz, and D. R. Musser, "The Design of Data Type Specifications," in *Current Trends in Programming Methodology*, Vol. IV, R. T. Yeh, ed., Prentice-Hall, 1978.

14. Guttag, J. V., and Horning, J. J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, 10, 27-52, 1978.

15. von Henke, F. W. and Luckham, D. C., "A Methodology for Verifying Programs," *Proceedings of 1975 International Conference on Reliable Software*, Los Angeles, April 1975, pp. 156-163.

16. Huet, G. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems," IRIA - LABORIA Report No. 250, Domaine de Voluceau, 78150 Rocquencourt, France.

17. Huet, G. and D. S. Lankford, "On the Uniform Halting Problem for Term Rewriting Systems," IRIA - LABORIA Report.

18. Knuth, D. E. and P. B. Bendix, "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, J. Leech, ed., Pergamon Press, New York, 1970, pp. 263-297.

19. Lankford D. S., *Canonical Inference*, University of Texas Automatic Theorem Proving Project Report ATP-32, December 1975.

20. Lankford, D. S. and A. M. Ballantyne, *Decision Procedures for Simple Equational Theories with Commutative-Associative Axioms: Complete Sets of Commutative-Associative Reductions*, University of Texas Automatic Theorem Proving Project Report ATP-39, August 1977.

21. Lipton, R. and Snyder, L., "On the Halting of Tree Replacement Systems," Conference on Theoretical Computer Science, University of Waterloo, 1977.

22. Musser, D. R., "A Data Type Verification System Based on Rewrite Rules," *Proceedings of the Sixth Texas Conference on Computing Systems*, Austin Texas, November 1977.

23. Musser, D. R., "Abstract Data Type Specification in the AFFIRM System," *Proceedings of the Specifications of Reliable Software Conference*, Boston, April 3-5, 1979, pp. 47-57. Also to appear in *IEEE Transactions on Software Engineering*.

24. Nakajima, R., "Sypes--Partial Types--for Program and Specification Structuring and a First Order System Iota Logic," Research Report No. 22, Institute of Informatics, University of Oslo, November 1977.

25. Nelson, G. and D. C. Oppen, "A Simplifier Based on Efficient Decision Algorithms," *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson Arizona, January 1978.

26. Spitzen, J., and B. Wegbreit, "The Verification and Synthesis of Data Structures," *Acta Informatica*, vol. 4, (1975), pp. 127-144.

27. Stickel, M. E. and G. E. Peterson, "Complete Sets of Reductions for Equational Theories with Complete Unification Algorithms," Department of Computer Sciences, University of Arizona, and Department of Mathematical Sciences, University of Missouri, September, 1977.

28. Suzuki, N., "Verifying Programs by Algebraic and Logical Reduction," *Proceedings of 1975 International Conference on Reliable Software*, Los Angeles, April 1975, pp. 473-481.

29. Zilles, S.N., *An Introduction to Data Algebra*, Draft Working Paper, IBM San Jose Research Lab., Sept. 1975.