PUC

Series: Monografias em Ciência da Computação

Nº 1/81

COMPARING ABSTRACT DATA TYPE SPECIFICATIONS VIA THEIR NORMAL FORMS.

by

Jean Luc Remy

Paulo A.S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453

RIO DE JANEIRO – BRASIL

Series: Monografias em Ciência da Computação
Nº 1/81
Series Editor: Marco A. Casanova March, 1981

COMPARING ABSTRACT DATA TYPE SPECIFICATIONS VIA THEIR NORMAL FORMS

by

Jean Luc Remy*
Paulo A.S. Veloso**

* Centre de Recherche en Informatique de Nancy, Université de Nancy, 54037 Nancy, France.

** Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 22453 Rio de Janeiro, RJ, Brasil;
sponsored in part by the French Ministry for Foreing Affairs and the Brazilian National Council for Scientific and Technological Development.

ABSTRACT:

A simple technique is presented for verifying that two abstract data type specifications are equivalent in that they have isomorphic initial algebras. The method uses normal forms to attemp reducing the number of equations to be checked. It is applied to a simple example and some extensions and related problems are also discussed.

KEY WORDS:

Abstract data type, formal specification, rewriting system, normal from, initial algebra, equivalence proof.

RESUMO:

Apresenta-se uma técnica simples para se verificar que duas especificações de tipos abstratos de dados são equivalentes no sentido de terem álgebras iniciais isomorfas. O método usa formas normais para tentar diminuir o número de equações a serem testadas, sendo aplicado a um exemplo simples. Além disso, discutem-se algumas extensões a problemas relacionados.

PALAVRAS CHAVES:

Tipo abstrato de dados, especificação formal, sistema de re-escrita, forma normal, álgebra inicial, prova de equivalência.

1 - Introduction

We propose some improvements on a methodology to check the equivalence of two given abstract data type specifications. The classical method consists in establishing an isomorphism between the congruence classes of the two specifications or, equivalently, proving that all the rules of each specification are theorems of the other. The main problem with this simple minded approach is the high number of the theorems to be verified. In order to reduce this number our method uses criteria about normal forms.

The need to compare specifications appears frequently when dealing with abstract data types, as one often tries to improve a given specification aiming at clarity, efficient implementations, etc.

The structure of the paper is as follows. First we present a simple example of two alledgedly equivalent specifications for the same data type. Then we prove the main result, which is in the sequel applied to the example and to enrichment of it. Thereafter we present some extensions and conclude with some comments on other applications of the method and related problems.

2 - An example

Consider the simple case of linear lists, consisting of two sorts Atom and List. Intuitively the elements of the sort List are finite, possibly empty, sequences of atoms. We shall consider the following operations (together with an intuitive description of their intended meanings):

Null (the empty list), Unit (which makes a list consisting solely of a, out of atom a), Cons (which adds atom a in front of list ℓ) and Append (Append (ℓ,ℓ') being the result of appending ℓ' after ℓ).

We shall present two algebraic specifications for this data type. The main part of each one is a set of rewriting rules, which defines a set of normal forms and the effect of each operation on them. These specifications can be regarded as arising from different manners of constructing lists.

One way of describing these lists is as follows. First we have the empty list, denoted by Null. Then we have the lists of length one, denoted by Unit (a) for a in Atom. Finally, we can obtain longer lists by repeated applications of Append. But, this operation is intended to be associative and to have Null is its identity. So in order to have unique names for the lists, we restrict the applications of Append. Thus we arrive at the following set of normal forms

$$F_1 = \{\text{Null}\} \cup \{\text{Unit}\ (a)\ /\ a \in \text{Atom}\} \cup$$
$$\cup \{\text{Append}\ (\text{Unit}\ (a_1), \ldots, \text{Append}(\text{Unit}\ (a_{n-1}), \text{Unit}(a_n)) \ldots) /\ a_1, \ldots, a_n \in \text{Atom},\ n>1\}$$

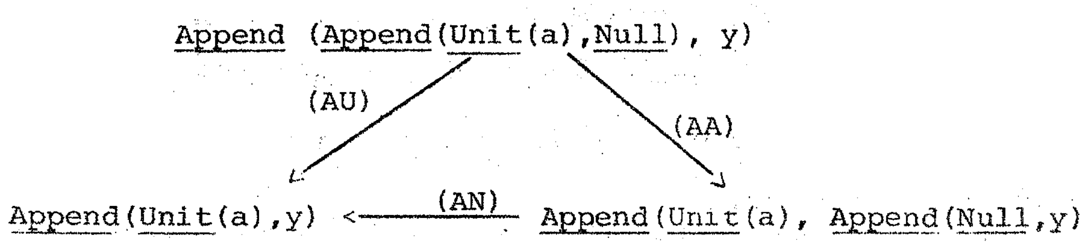
In other words, a normal form in F_1 is either Null, or Unit(a), for a in Atom, or else Append (Unit(a), f), for a in Atom and f in with f $\neq$ Null. So, the operation symbols occurring in the terms of F_1 are Null, Unit and Append. A specification Σ_1 for List (Atom) based on these normal forms appears in Fig. 1.

Formally, the normal forms involve not arbitrary a's in Atom but their normal forms. However, we leave this implicit by taking the same normal forms for the sort Atom in both specifications.

Now, one can verify that the set R_1 of rules of Σ_1 has the properties of finite termination and confluence (a Church - Rosser property), which ensure that each term reduces to a unique normal form [8].

Finite termination can be shown by applying Musser's criterion [11] for proving the termination of rewriting systems obtained by iterated enrichments. The main point is connected to the associativity of Append: in rule (AA) the first arguments of Append in the righthand side (namely, Unit (a) and x) are simpler than the corresponding one in the lefthand side (namely Append (Unit(a), x)).

Confluence can be verified by showing local confluence on the critical pairs, according to the Knuth-Bendix criterion [9]. In this case we have



One can also verify that the irreducible terms of Σ_1

are exactly those in F_1. Indeed, every normal form in F_1 is clearly irreducible and, by induction, every $t \notin F_1$ is shown reducible.

On the other hand, there is another way of constructing lists, which gives origin to a different set F_2 of unique names and a specification with other constructors. Namely, a list is denoted by either Null or else Cons(a,g) for a in Atom and g in F_2 . So

F_2 = { Cons(a_1,..., Cons(a_n, Null)...) /a_1,...,$a_n \in$ Atom, n≥0 }

where we agree that the case n=0 corresponds to Null. Now Append and Unit become internal operations.

A specification Σ_2 for List (Atom) corresponding to these normal forms is shown in Fig. 2. Notice that this set R_2 of rules has no rule between constructors. Moreover, it is quite simple to check that it is finitely terminating and confluent.

Now, a question arising naturally is whether Σ_1 and Σ_2 are indeed equivalent, in that they specify the same data type. One way to show their equivalence is by verifying that they have the same theorems. This is equivalent, as they are confluent and and finitely terminating, to establishing an isomorphism between their initial algebras, which gives a bijection between their normal forms. In the next section we shall show how this idea enables us to reduce the number of theorems to be checked.

```
Type    List  ( Atom )

  Sorts    List  ,  Atom

  Operations    with  a:  Atom ;  x,y : List

      { constructors  }
                              Null   : List
                             Unit(a): List
                         Append(x,y)  : List

          { internal }
                         Cons(a,y)  :  List

  Rules          for   each  a: Atom   ; x,y : List
          { between  constructors }

  (AN)         Append(Null,y) → y
  (AU)         Append(Unit(a), Null) → Unit(a)
  (AA)         Append(Append(Unit(a),x),y)→
                     →   Append(Unit(a),Append(x,y))

          { defining internal operation }
  (C )        Cons(a,y) → Append(Unit(a),y)

end of type
```

Fig. 1 : Specification Σ_1

```
Type    List    (Atom)

    Sorts    List,  Atom

    Operations  with   a:  Atom ;  x,y : List

       { constructors }
                  Null  :  List
              Cons(a,y)  :  List
        { internal }

                  Unit(a): List

              Append(x,y)  : List

     Rules   for each a   : Atom ; x,y : List

        { defining internal operations }

     (U)     Unit(a) → Cons (a,Null)
     (AN)    Append(Null,y) → y
     (AC)    Append(Cons(a,x),y) → Cons(a,Append(x,y))

end of type
```

Fig. 2 : Specification Σ_2

3 – The main result

A specification Σ consists of a set S of sorts, a set O of operation (symbols) together with their profiles, and a set R of term rewriting rules, which we assume confluent and finitely terminating.

Denote by T(X) (respectively T) the set of terms with variables in X (respectively variable-free terms) and by $\mathcal{T}(X)$ (respectively $\mathcal{T}$) the term algebra on T(X) (respectively T). On T(X), let $t \sim t'$ iff both reduce to a common $t'' \in T(X)$. Let the congruence on $\mathcal{T}(X)$ generated by $\sim$ be denoted by $\equiv$, the same symbol being used for its restriction to $\mathcal{T}$. The data type specified by Σ is $I(\Sigma) = \mathcal{T}/_{\equiv}$. Call F the set of irreducible terms of T and notice that each $t \in T$ reduces to a unique $f \in F$ and that for $u, v \in T(\{x\})$, $u \equiv v$ if $u[f/x] \equiv v[f/x]$ for all $f \in F$.

We shall be considering two specifications Σ_1 and Σ_2, both with the same sets S of sorts and O of operations. Let R_j, F_j, $\equiv_j$ denote, respectively, the set of rules, set of normal forms and the equality of Σ_j, for $j = 1,2$.

Theorem. Let Σ_1 and Σ_2 be as above.

1. If for each rule $u \to v$ of R_2 we have $u \equiv_1 v$ then $\equiv_2 \subseteq \equiv_1$.
2. If, in addition, for each normal form $g \in F_2$ there exists a normal form $f \in F_1$ with $g \equiv_2 f$ then $\equiv_2$ and $\equiv_1$ coincide on $\mathcal{T}$.

Proof

1. Clear from the definitions of $\sim$ and $\equiv$.
2. Consider t, t' in T with $t \equiv_1 t'$ and let $g, g' \in F_2$ be their normal forms, so that $t \equiv_2 g$ and $t' \equiv_2 g'$. By assumption ,

there exist $f, f' \in F_1$ with $g \equiv_2 f$ and $g' \equiv_2 f'$.
Thus $t \equiv_2 f$ and $t' \equiv_2 f'$, whence by 1, $t \equiv_1 f$ and $t' \equiv_1 f'$. So, since $t \equiv_1 t'$ the same holds for f and f' and $f = f'$, as both of them are in F_1. Therefore, $t \equiv_2 f = f' \equiv_2 t'$ QED

The idea behind the theorem is very simply described in terms of the discussion at the and of section 2: conditions 1 and 2 have the effect of guaranteeing the bijection between normal forms corresponding to the isomorphism of their initial algebras.

The conditions 1 and 2 of the theorem are clearly necessary for the equivalence of Σ_2, thus we have a test for equivalence. Not only does the failure of either of these conditions imply non-equivalence, but, more important, it helps pinpointing the trouble spots and may suggest modifications of Σ_1 or Σ_2 in order to achieve equivalence.

4 - Application and a criterion

We have, in section 2, two specifications Σ_1 and Σ_2 allegedly for the same data type. They have the same sets of sorts and of operations and are finitely terminating and confluent. So, we can apply our theorem.

1. We have to check that each rule of R_2 is a theorem of Σ_1. The case of (AN) is trivial and for (U) we have in Σ_1

   $$\underline{\text{Cons}}(a,\underline{\text{Null}}) \xrightarrow{(C)} \underline{\text{Append}}\ (\underline{\text{Unit}}(a),\ \underline{\text{Null}}) \xrightarrow{(AN)} \underline{\text{Unit}}(a)$$

   As for (AC), we have in Σ_1

   $$\begin{array}{ccc} \underline{\text{Append}}\ (\underline{\text{Cons}}(a,x),y) & \xrightarrow{(C)} & \underline{\text{Append}}\ (\underline{\text{Append}}(\underline{\text{Unit}}(a),x),y) \\ & & \downarrow (AA) \\ \underline{\text{Cons}}(a,\ \underline{\text{Append}}(x,y)) & \xrightarrow{(C)} & \underline{\text{Append}}(\underline{\text{Unit}}(a),\ \underline{\text{Append}}(x,y)) \end{array}$$

2. Now we have to check that each $g \in F_2$ is R_2-equal to some $f \in F_1$. The case of Null is obvious, as $\underline{\text{Null}} \in F_1$. For $g = \underline{\text{Cons}}(a,\underline{\text{Null}})$ we can take $f = \underline{\text{Unit}}\ (a)$, since in Σ_2 $f \xrightarrow{(U)} g$. Now, for $g = \underline{\text{Cons}}(a,g')$ with $g' \in F_2$ and $g' \neq \underline{\text{Null}}$, assume that we have $f' \in F_1$ such that $g' \equiv_2 f'$ and take $f = \underline{\text{Append}}(\underline{\text{Unit}}(a),\ f')$. Then, in Σ_2

   $$\underline{\text{Append}}(\underline{\text{Unit}}(a),\ f') \xrightarrow{(U)} \underline{\text{Append}}\ (\underline{\text{Cons}}(a,\underline{\text{Null}}),\ f')$$

   $$\xrightarrow{(AC)} \underline{\text{Cons}}(a\ ,\ \underline{\text{Append}}(\underline{\text{Null}},\ f')) \xrightarrow{(AN)} \underline{\text{Cons}}(a,f')$$

   whence $f \equiv_2 \underline{\text{Cons}}(a,f') \equiv_2 \underline{\text{Cons}}(a,g')$.

   Therefore, by our theorem. we can conclude $I(\Sigma_1) \cong I(\Sigma_2)$

Let us examine more carefully what was involved in checking that for each $g \in F_2$ we have $f \in F_1$ with $g \equiv_2 f$.

First, as $\equiv_2 \subseteq \equiv_1$, f is necessarily the Σ_1 - reduction of g. Second, we had to eliminate Cons from g , for it is a Σ_2-constructor but not a Σ_1-constructor. Finally, we can derive from R_1 two rules describing how in Σ_1 the internal operation Cons transforms the normal forms. Namely

(C1) Cons(a,Null) $\xrightarrow{*}$ Unit(a) ,

(C2) Cons(a,f) $\xrightarrow{*}$ Append(Unit(a),f)

Notice that the righthand side of (C2) is in F_1 if $f \in F_1$ and $f \neq$ Null and that repeated applications of (C1) and (C2) give the Σ_1-reduction of each normal form in F_2. So, all we had to do was checking that (C1) and (C2) are theorems of R_2.

We now state a useful criterion generalizing these ideas. It is based on the partioning of the operations of a data type into constructors and internal operations, according to their occurring or not in a normal form. The extra assumptions are frequently easy to verify when the normal forms have recursive definitions.

Proposition . For j = 1,2 let Σ_j be as before, C_j the set of operations occurring in the normal forms in F_j and $I_j = O - C_j$. Assume that

a) for each $g \in F_2$ without operations from I_1 there exists $f \in F_1$ with $g \equiv_2 f$.

b) for each $o \in C_2 \cap I_1$ and $f_1,\dots,f_k \in F_1$, we have $o(f_1,\dots,f_k) \equiv_2 f$ with $f \in F_1$.

Then for each $g \in F_2$ there exists $f \in F_1$ with $g \equiv_2 f$.

<u>Proof</u> by induction on the number n of occurrences of operations of $C_2 \cap I_1$ in $g \in F_2$.

Case n=0 follows from assumption (a).

Case n>0, let g' be a subterm of f of the form $o(g_1', \ldots, g_k')$ with $o \in C_2 \cap I_1$. Then, for $i = 1, \ldots, k$, $g_i' \in F_2$ and by induction we have $f_i' \in F_1$ with $g_i' \equiv_2 f_i'$. Now, from (b) we have $f' \in F_1$ with $o(f_1', \ldots, f_k') \equiv_2 f'$, whence $g' \equiv_2 f'$. The term $\bar{g}$ obtained from g by replacing g' by f' to has fewer occurrences of operations of $C_2 \cap I_1$ than g, so by induction, $\bar{g} \equiv_2 f$ for some $f \in F_1$. Hence $g \equiv_2 \bar{g} \equiv_2 f$. QED

5 - Extension to parametrized specifications

In order to illustrate more clearly the advantages of this method, let us consider the data type List(Atom,Bool) obtained by enriching List(Atom) with a Boolean sort Bool, with Boolean constants True and False and the conditional operation If - then - else, together with the (external) operation

Equal: List × List → Bool (to test equality of lists) and

Same: Atom × Atom → Bool (checking equality of atoms).

We can decompose a specification of List(Atom, Bool) into two parts:

- a parameter specification of the sorts Atom and Bool, including some rules for defining the operations If-then-else and Same;
- a proper part consisting of the rules given in section 2 together with a set of rules enabling the reduction of each term of the form Equal(t,t') either to True or to False.

Consider the set of rules R_1' obtained by adding to R_1 the 11 rules below, where x,y: List; a,b,c: Atom

(E1) Equal (Null,Null) → True

(E2) Equal (Null,Unit(b)) → False

(E3) Equal (Null,Append(Unit(b),y)) → False

(E4) Equal (Unit(a),Null) → False

(E5) Equal (Unit(a),Unit(b)) → Same (a,b)

(E6) Equal (Unit(a),Append(Unit(b),Unit(c))) → False

(E7) Equal (Unit(a),Append(Unit(b),Append(Unit(c),y))) → False

(E8) Equal (Append(Unit(a),x), Null) → False

(E9) Equal (Append(Unit(a), Unit(b)), Unit(c)) → False

(E10) Equal (Append(Unit(a), Append(Unit(b),x)), Unit(c)) → False

(E11) Equal (Append(Unit(a),x), Append(Unit)(b), y)) →

→ If Same(a,b) then Equal (x,y) else False

It is tedious enough to write down this set of rules based on the recursive definition of the normal forms in F_1. Notice, in particular, that one cannot merge (E6) and (E7) into a single rule with lefthand side Equal(Unit(a), Append(Unit(b),y)). Like-wise for (E9) and (E10).

We now have a specification Σ_1' for List.

Had we used the set F_2 of normal forms we would have a specification Σ_2' with, set of rules R_2' consisting of R_2 together with the following 4 rules defining the external operation Equal

(ENN) Equal (Null,Null) → True

(ENC) Equal (Null,Cons(b,y)) → False

(ECN) Equal (Cons(a,x), Null)→ False

(ECC) Equal (Cons(a,x), Cons(b,y)) →

→ If Same(a,b) then Equal(x,y) else False

where a,b : Atom; x,y : List

The labor-saving fact here is this:

Having shown the equivalence of Σ_1 and Σ_2, it suffices to check that the above 4 new rules of R_2' are theorems of R_1 in order to conclude the equivalence of Σ_1' and Σ_2'.

Notice that this method reduces our job of checking the equivalence of Σ_1' and Σ_2' to

(i) proving all the 3+4 rules of Σ_2' in Σ_1';

(ii) proving 2 derived rules of Σ_1' in Σ_2' .

The more naive and symmetrical approach would, instead of (ii), involve

(ii') proving all the 4+11 rules of Σ_1 in Σ_2 .

So, we really cut down the number of theorems to be verified from 22 to 9.

Before presenting the result justifying the conclusion let us finish the verification of the equivalence of Σ_1' and Σ_2'. We have to check that the 4 rules in $R_2'-R_2$ are theorems of R_1'. For (ENN) it is trivial, and for (ENC) and (ECN) we use (C) and (E3), respectively (E8). Finally for (ECC)we have in R_1'

Equal(Cons(a,x), (Cons(b,y))

(C) ↓ ↓ (C)

Equal(Append(Unit)(a), x) Append(Unit)(b),y))

↓ (E11)

If Same(a,b) then Equal(x,y) else False

We shall now consider the extension of the theorem of section 3. First, we consider a parametrized specification Σ as consisting of

- a parameter specification $\Sigma_p = <S_p, O_p, R_p>$, forming a confluent and finitely terminating rewriting system;

- a designated sort s (sort of interest);

- a set O of operation (symbols) together with their profiles, each $o \in O$ having at least one argument or its result is S ;

a set R of rewriting rules such that $R \cup R_p$ is confluent and finitely terminating, and R is consistent and sufficiently complete (with respect to Σ_p) in the sense explained below.

Call F_p the set of normal forms of R_p and F^p the set of normal forms of $R \cup R_p$ for the sorts in S_p. Then R is consistent (with respect to Σ_p) iff $F_p \subseteq F^p$, and R is sufficiently complete (with respect to Σ_p) iff $F^p \subseteq F_p$.

We also recall that for a parametrized specification Σ as above with parameter specification Σ_p, the reduct of the data type $I(\Sigma)$ to the sorts in S_p and operations in O_p is isomorphic to $I(\Sigma_p)$ (see., e.g. [4]).

Now consider two parametrized specifications Σ_j with the same parameter specification Σ_p, same sort of interest s and same set O of operations, and let F_j be the set of normal forms of Σ_j for the sort s, for $j = 1,2$.

<u>Theorem</u> Let Σ_1 and Σ_2 be as above and assume

1. for each rule $u \to v$ of R_2 $u \equiv_1 v$;
2. for each $g \in F_2$ there exists $f \in F_1$ with $g \equiv_2 f$.

Then Σ_1 and Σ_2 are equivalent, i.e.

$I(\Sigma_1) \cong I(\Sigma_2)$

<u>Proof</u>

Similarly to the proof in section 3, conditions 1 and 2 ensure that the restrictions of $\equiv_1$ and $\equiv_2$ to the sort s coincide. For a sort in S_p condition 1 gives $\equiv_2 \subseteq \equiv_1$, i.e. each $\equiv_2$ - class is included in a $\equiv_1$-class. However, due to the consistency and sufficient completeness of Σ_j, each $\equiv_j$ class intersects a unique

$\equiv_p$-class of the parameter specification, for $j = 1,2$. Therefore, $\equiv_1$ and $\equiv_2$ coincide on parameter sorts, as well. QED

6 - Conclusion

We have shown that two abstract data type specifications can be be proven equivalently more simply than by proving all the rules of each one to be theorems of the other. In our example we had to verify only 9 rules instead of 22.

These ideas appear to be connected to results Huet and Hulot [7], Goguen [3] and Musser [11]. They propose methods for proving indictive properties without induction and use for this the Knuth-Bendix completion algorithm [9].

The need to compare specifications appears naturally when transforming specifications for a data type trying to obtain one that leads to more efficient implementations, for instance. Such is the case of a more complex example with two independent constructors developed in [2]. There, starting from a specification which considers the data type pointed lists of [1] as consisting of pairs (list, integer), another pair of constructors is obtained, which leads to a simpler specification.

There is yet another interesting aspect to the idea of changing constructors, namely the so-called hidden operations [6]. In our example of section 2, call Σ_0 the specification obtained from Σ_1 by removing the operation Cons and the rule (C). We may get Σ_1 back by enriching Σ_0 with Cons. On the other hand, call Σ_3 the specification obtained from Σ_2 by hiding the constructor Cons. Then $I(\Sigma_3)$ is the reduct of $I(\Sigma_2)$ to the non-hidden operations. Hence $I(\Sigma_0) \cong I(\Sigma_3)$ and thus the specifications Σ_3 with hidden operation is equivalent to Σ_0, with the advantage of being simpler.

Finally let us notice that our results rely on the confluence, finite termination and sufficient completeness of the specifications. It would be desirable to have more systematic methods to verify these properties. Also useful would be a methodology to derive the rules of Σ_2 from those of Σ_1, given the constructor sets (cf. the derived rules in section 4).

Acknowledgements

The authors are grateful to Christine Choppy and Pierre Lescanne for discussions about equivalence between specifications. The first author is especially, indebted to Pierre Lescanne, whose thesis [10] inspired him in many aspects.

References

[1] - W. Bartussek and D. Parnas, Using traces to write abstract specifications for software modules, Research Report, Department of Computer Science, University of North Carolina, NC (1977).

[2] - C. Choppy, P. Lescanne and J.L.Remy, Improving abstract data type specifications by appropriate choice of constructors. Proc. Intern. Workshop on Program Construction, Bonas, France (1980)

[3] - J. A. Goguen, How to prove algebraic inductive hypotheses without induction, with application to the correctness of data type implementation, Proc. 5th Conference on Automated Deduction, Les Arcs, France (1980).

[4] - J. A. Goguen, J. W. Thatcher and E. G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. T. Yeh, ed., Current Trends in Programming Methodology, vol. IV (Prentice hall, Englewood Cliffs, NY, 1977) 80-149.

[5] - J. V. Guttag and J.J. Horning, The algebraic specification of abstract data types, Acta Informat. 10 (1978) 27-92.

[6] - J. V. Guttag, E. Horowitz and D. R. Musser, The design of data type specifications, in: R.T. Yeh, ed., Current Trends in Programming Methodology, vol. IV (Prentice-Hall; Englewood Cliffs, NY 1977) 60-79.

[7] - G. Huet and J. M. Hullot, Proofs by induction in equational theories with constructors , Research Report nº 28, INRIA , Rocquencourt, France (1980)

[8] - G. Huet and D. C. Oppen, Equations and rewrite rules : a survey, Research Report Nº STAN-CS. 80-785, Computer Science Department, Stanford University, Palo Alto, CA (1980). Also in: R. Book, ed., Formal Languages: perspectives and open problems, Academic Press, New York , NY (1980).

[9] - D.E. Knuth and P. Bendix, Simple word problems in universal algebra, in: J. Leech, ed., Computacional problems in abstract algebra, Pergamon Press, London(1970) 263-297 .

[10] - P. Lescanne, Etude algebrique et relationelle des types abstraits et de leurs representations, Inst. National Polytechnique de Lorraine, thèse d'Etat, Nancy (1979) .

[11] - D.R. Musser, Convergent sets of rewrite rules for abstract data types, Research Report, Information Sciences Institute, University of Southern California (1978) .

[12] - D.R. Musser, On proving inductive properties of abstract data types, Proc. 7th Annual ACM Symposium on Principles of Programming Languages (1980) 154-162.

JLR/PASV/jrp.